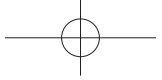




O'REILLY®

SVG动画

SVG Animations

[美]Sarah Drasner 著
大漠 姜天意 欧阳湘粤 田淮仁 张耀春 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

在制作Web动画效果时，使用SVG制作动画是我们应该掌握的技能之一。本书详细介绍了SVG的基础知识、如何使用SVG制作动画、制作SVG动画的工具及相关的JavaScript库。除此之外，本书也探讨了SVG还能做的一些十分有趣的事情，比如数据可视化、可伸缩的矢量图、响应式设计等。Sarah为广大读者提供了一本非常优秀的书籍，可帮助读者系统地掌握SVG和SVG制作动画相关的技术知识。如果你想掌握这项技术，那么本书是值得你花时间去阅读和研究的一本书。

版权贸易合同登记号　图字：01-2017-3324

图书在版编目（CIP）数据

SVG动画/（美）莎拉·德拉斯纳（Sarah Drasner）著；大漠等译. —北京：电子工业出版社，2018.5
书名原文：SVG Animations
ISBN 978-7-121-33790-1

Ⅰ. ①S… Ⅱ. ①莎… ②大… Ⅲ. ①图形软件 Ⅳ. ①TP391.412

中国版本图书馆CIP数据核字（2018）第039911号

策划编辑：张春雨
责任编辑：刘　舫
封面设计：Karen Montgomery　张　健
印　　刷：北京天宇星印刷厂
装　　订：北京天宇星印刷厂
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　　印张：13.75　　字数：301千字
版　　次：2018年5月第1版
印　　次：2018年5月第1次印刷
定　　价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。
质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。
本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

谨以此书献给 *Dizzy*！

本书赢得的赞誉

很少有人像 Sarah 那样热衷于 Web 动画，她的新书就是一个宝库。如果你想在 Web 上使用 SVG 动画，那么必须阅读这本书。

—**Jack Doyle**，*GreenSock*

Sarah Drasner 写的动画总是令人愉悦，流畅而优雅。她不仅是一位优秀的动画设计师，她还解释了为什么以及如何使用这些工具来创造你想要的动画。她通过简单明了的方式指导读者理解重要的概念，并为读者推荐能用于跨浏览器制作动效的库。即使你对 SVG 和动画原理有所了解，也不会后悔拥有这本书，因为书中的内容能让你变得更强大。

—**Chris Lilley**，*SVG* 之父

SVG 动画对于任何使用 SVG 的人来说都是必须掌握的。Sarah Drasner 把她所知道的动画知识点整合在一起，向读者展示了如何在动画中做出最好的选择，以及如何用最专业的技术来完成动画。

—**Val Head**，设计界知名动画设计师

Sarah Drasner 既是一位令人难以置信的艺术家，也是一位务实、注重细节的 Web 开发者。SVG 动画为 Web 上的矢量动画提供了实用的解决方案，并且通过一系列的工具帮助你不让技术限制你的创造力。

—**Amelia Bellamy-Royds**，*SVG Colors, Patterns & Gradient*、*SVG Essentials* (second edition)、*SVG Text Layout* 和 *Using SVG with CSS3 and HTML5* 的作者

译者序

众所周知，SVG（可缩放矢量图）对于 Web 而言并不是一个全新的课题，只不过近些年 SVG 在 Web 页面或者 Web 应用程序中使用的频率越来越高。这次非常荣幸能和社区的小伙伴一起参与翻译 Sarah 的著作。

SVG 自发明以来沉寂了非常长的一段时间，由于浏览器厂商对 SVG 技术支持不够强大，所以在 Web 中的使用机会并不多，而且其发展也是曲折的。不过值得庆幸的是，SVG 以它优秀的兼容性、可适配性在开源社区重新拥有了重要的地位，而且近一两年来，使用 SVG 的场景也越来越多。

相对而言，SVG 具有如下一些独特的优点。

- **数据可视化**：真实的数据直观地表达了人们想要表达的想法，并且易于复杂思想的交流和展现，而 SVG 对于数据可视化的展示来说具有一定的优势和可扩展性。
- **响应式**：对于设备品类繁多的今天，如果要让一张图片能很好地兼容各种设备平台，那么 SVG 是最佳选择之一。因为 SVG 是矢量图，其具有独特的可伸缩性，能很好地适应各种屏幕尺寸的设备。
- **性能表现**：如果在 Web 应用或 Web 页面中正确使用 SVG，可以减少网站加载的资源，特别是在响应式 Web 应用中，这个特性显得更为显著。
- **可操作的 DOM 结构**：对于开发者而言，SVG 还有另一个有吸引力的特性，那就是它类似于 HTML，具有可操作的 DOM。这意味着用户可以使用代码直接实现想要的矢量图效果，而且还能为屏幕阅读器提供更多的信息，并且可以很好地通过给 DOM 添加适合的动画，让 SVG 炫动起来。

本书由始至终都贯穿着 SVG 的相关知识。我们将讨论 SVG DOM 的基础知识，它结构简单，能让人感觉代码浅显易懂。还将讨论 SVG 性能方面的知识，将学到如何精简 SVG 文件的体积和结构，借此避免网站性能由于 SVG 的原因大打折扣。必不可少的，我们还会讨论如何通过 CSS 和 JavaScript 制作 SVG 动画，来完成优美且有趣的动画效

果。如果你想了解如何设计 SVG（编码 + 优化），那么请阅读本书的前半部分。本书的后半部分将更加适合 JavaScript 开发者学习如何使用 JavaScript 控制 SVG 动画。SVG 这一技术将设计和开发的世界交融起来，所以本书对这两个方面都会涉及。

通过学习本书，读者能比较全面和深入地了解 SVG 的基础知识，特别是 SVG 动画的制作。为了能更好地帮助读者快速有效地使用 SVG 制作动效，Sarah 还介绍了各种工具和原生 JavaScript 库。

能将这本书翻译出来，非常感谢社区的同学（@ 姜天意、@ 欧阳湘粤、@ 田淮仁和 @ 张耀春）花费宝贵的时间参与翻译，同时要特别感谢博文视点的张春雨、刘舫以及其他工作人员的帮助，在此一并表示由衷的感谢。

本书主要由我和社区的同学共同翻译，虽然我们经常参与社区前端技术文档的翻译，但翻译 SVG 动画相关的书籍还是第一次，因此全书难免存在一些错误或者不当之处，敬请广大读者批评指正。我们非常愿意通过电子邮件（w3cplus@hotmail.com）与各位同行探讨有关于 SVG 或 SVG 动画相关的技术问题。

大漠

2018 年 3 月于杭州

目录

序

你有没有过这样的经历，新学了一个单词，然后立刻想找一个适合的情景使用一下那个新学会的词？学习 SVG 就会产生这样的感觉。在隐喻的层面上来说，SVG 好比是你的工具箱中一直缺少的那个趁手的工具。

对设计师和开发人员来说，现在已经到了深入学习 SVG 的时候了。我可以告诉你，我几乎在每天的工作中都会使用到 SVG。不是因为我能把 SVG 运用到我的项目中，而是因为它对我的工作来说是一个合适的工具。在你阅读本书后，SVG 也会成为你工作中一个十分重要的工具，你也会经常使用它。

当需要在任何像素密度的屏幕上清晰地显示一个图标的时候，或许你会想到 SVG。当需要一个图标系统、图表或矢量背景图时，你也可能会想到 SVG。当想要制作动画时，你可能会想到你手中拿的这本书，也会想到 SVG。

SVG 是很适合制作动画的，它是 Web 上最强大的动画制作工具之一。原因之一是因为 SVG 是由数字组成的，SVG 本质上是利用几何图形进行绘制的。在 Web 上，数字很容易被操作，也非常直观。或许你可能知道，让一个元素淡出——这是一个基本的动画效果——就是把透明度从 1 变成 0 的一个动画效果。也可以将改变一个圆的半径或矩形的坐标做成动画效果，或者用一个路径上的点来做一些动画效果。

SVG 动画如此引人注目的另一个原因是有多种方法可以实现，有多种技术供你选择。你要知道的是做出怎样的选择，这时需要一些知识和思考。幸运的是，这本书能告诉你怎样做出合理的选择。

本书作者 Sarah 是带领大家在 SVG 的世界中遨游的终极导游。她不仅是一位有经验的技术专家，而且还是一位有成就的设计专家和走在前沿的开发者。多年来，她一直将自己的 SVG 艺术带到生活中，并把自己所知道的工具都运用到实际工作中，而且她知道如何去解释这一切。



我和 Sarah 一起工作过，并把她的 SVG 动画制作知识运用到我的项目中。如果你在想“我已经是一位前端开发人员，而且对 SVG 还没有很好的认识”，那么请阅读这本书，它可以帮你掌握还没有掌握的知识，而且还能助你成为一位 SVG 专家。

Chris Coyier

前言

艺术和代码相交，诞生了 SVG 动画

人们常常开玩笑地形容，从事 SVG（可缩放矢量图）的人，一定是一个“考古学家”，并且是和 SVG 一样有趣的人。这个比喻比较恰当。先前的 SVG 技术，由于缺乏浏览器厂商的支持和人们对这种技术的理解，曲折地发展了一段时间，但是现在它以优秀的兼容性重树威信，在开源社区拥有了很高的地位，以下是 SVG 的一些优点。

数据可视化

SVG 允许人们通过真实的数据直观地表达想法，并着重于复杂的思想交流展现。

设备响应式

世界上需要兼容的仪器、网页视窗、视网膜设备有成百上千种，SVG 有能力通过使用一张矢量缩放图来兼容各种设备，让你的资源加载列表变得清爽简单。SVG 是响应式兼容配置的游戏规则改变者。

性能表现

如果我们正确地构建 SVG，并减少路径点和简化图形细节（优化手段），SVG 文件的大小会十分轻量，位图无法与其竞争。通过正确构建 SVG，可以让我们的网站在加载资源时变得快捷可用。

可操作的 DOM 结构

对于开发者来说，SVG 拥有可操作的 DOM 结构这一点，起初并不起眼，但实际上这个特点对于开发却是极其方便的。拥有一体化 DOM 结构的 SVG，意味着你的代码可以为屏幕阅读器提供更多的信息，并且让图形代码看起来非常容易理解。你还可以正确地找到图形片段的内部节点，为这些小的 DOM 片段同时添加动画和操作。在本书中，你会发现这个特性真的很强，它能让你远离那些将人逼疯的有关 `z-index` 和 `position` 相关操作的设置。

我们可以流畅自由地使用 SVG 制作动画，但这仅仅触及了这项技术强大潜力的表面。作为一名开发者，当看到灵活的响应式媒体动画组件和 Snap 动画时，你一定会极其兴奋。SVG 还有令人称奇的一点就是可以使用数学方法绘制图案。

本书由始至终都贯穿着 SVG 的相关知识。谈到 SVG，我们就要讨论与 SVG DOM 相关的基础知识，它结构简单，工作起来让人觉得代码浅显易懂。我们还将讨论 SVG 性能方面的设计，将学会如何精简 SVG 文件的体积和结构，借此避免由于 SVG 的原因使网站性能大打折扣。必不可少的，我们还将讨论如何通过 CSS 操控 SVG 进行动画制作，学习一些相关理论，然后学习使用 JavaScript 深入 SVG 动画制作，以完成优美且有趣的各种动画效果。如果想了解如何设计 SVG（编码 + 优化），那么你可以详细阅读此书的前半部分。本书的后半部分更加适合 JavaScript 开发者学习如何使用 JS 控制 SVG 动画，SVG 这一技术将设计和开发的世界融合起来，所以本书两个方面都会涉及。我建议读者首先阅读第 1 章，因为它是理解后续章节的基础。

使用 SVG 制作动画，是我的互联网从业生涯中最激动人心的一部分：它激发出了我关于代码性能、代码易用性、设计美感以及创造性的潜力，让我学会避免了一些丑陋的代码以及图片黑科技（hack 技术），让我在需要的时候能够对页面进行响应式设计。这一技术还让我创作出了许多令人惊叹的数据可视化作品，帮助我方便地通过图表与用户交流、讲述原理，或者做一些优化用户交互体验的改进。

虽然我拥有 15 年以上的前端开发经验，但是学习 SVG 动画技术，激起了我的动画制作兴趣，为我开拓了新的开发领域。我希望你也能够喜欢这本书，并且为互联网创造出更多动态 SVG 作品。这是一个令人激动的时代，SVG 的潜力还未完全激发出来。目前，只有一小部分的 SVG 功能运用到 Web 设计和开发中。我期待看到你在读完此书之后，借助学到的知识所做的优雅作品。让我们一起推动互联网再一次的革新浪潮吧!

本书使用的约定

本书使用的约定如下。

斜体（*Italic*）

用于表明新术语、网址、电子邮件地址、文件名和文件扩展名。

等宽字体（`Constant width`）

用于程序清单，以及在段落中对变量、函数名、数据库、数据类型、环境变量、语句和关键字等程序元素的引用。

等宽加粗字体（**Constant width bold**）

用于显示命令或其他需要用户输入的文字。

等宽斜体字体（*Constant width italic*）

用于显示应该由用户提供或者根据上下文确定的值。

该图标表示提示或建议。

该图标表示一般性的注释。

该图标表示提醒或警告。

中文版书中切口以“▭”表示原书页码，便于读者与英文原版图书对照阅读，本书索引中所列的页码也为英文原版图书中的页码。

关于使用本书示例代码

我们感谢但不要求注明出处。出处的格式一般包括标题、作者、出版商和ISBN。示例如下：“*SVG Animations* by Sarah Drasner (O’Reilly). Copyright 2017 Sarah Drasner, 978-1-491-93970-3.”

如果你觉得示例代码的使用不合理或不符合以上的许可权限，请随时联系我们：*permissions@oreilly.com*。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者：

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网站，你可以在那里找到关于本书的相关信息，包括勘误列表、示例代码以及其他信息。本书的网站地址是：

http:/oreil.ly/2nouksg

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于我们的书籍、课程、会议和新闻的更多信息，请参阅我们的网站 *http://www.oreilly.com*。

在 Facebook 上找到我们：*http://facebook.com/oreilly*
在 Twitter 上关注我们：*http://twitter.com/oreillymedia*
在 YouTube 上观看我们：*http://www.youtube.com/oreillymedia*

致谢

首先我想感谢 Meg Foley，她是一名有着惊人工作能力的编辑。她对我的指导和关心，我感激不尽。可以说，没有她，就没有这本书。

其次，我想感谢那些技术校审者：Amelia Bellamy-Royds、Dudley Storey 和 Val Head，他们经过多番讨论最终帮助我确定了此书的脉络结构，感谢你们的辛苦付出。我还想感谢 GreenSock 组织的 Jack Doyle 和 Carl Schooff、React-Motion 的 Cheng Lou，以及 mo.js 的作者 Oleg Solomka。感谢他们审校了本书关于他们资源库的链接地址以及为互联网创造了如此令人赞叹的动画工具。感谢 SVG 发明者 Chris Lilley，他是一名值得尊敬的先驱！感谢他为本书写了如此精彩的序，同时，在技术文章的写作方面亦是我的良师益友。他还教会了我一些 CSS 技巧和其他知识。

最后，我想着重感谢 Dizzy Smith、Meagan French 和 Donna Ferriero，他们在我写作本书期间给予了极大的支持。特别是在我好高骛远的那段日子中，感谢你们对我的照顾以及陪我走过了一段又一段里程。你们是最棒的!

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- 提交勘误：你对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在购买电子书时，积分可用来抵扣相应金额）。
- 交流互动：在页面下方读者评论处留下你的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/33790

第 1 章

剖析SVG

可缩放矢量图（SVG）正日益成为互联网中一种较流行的图片格式，这种格式的优点可以概括为以下几点。

- SVG 图片是可缩放的，这对于市面上越来越多需要适配的屏幕视窗（viewport）有巨大的好处。我们借助 SVG 技术可以实现：只制作一张 SVG 图片，就可以对所有设备做到适配兼容，并进行缓存，下次访问时不需要再次请求。而对比 CSS 技术来说，我们可能需要用到 `srcset` 或根据不同的视窗所需的图片来配置 `@media` 等方案来解决适配的问题。SVG 避免了所有的额外工作。
- 矢量图（相对位图来说）意味着，由于使用了数学方法绘制，图片文件有着更好的质量和更小的体积。

SVG 是一种 XML 格式的文件，我们可以使用这种格式简洁地描述图形、线和文本，并且可以同时使用 DOM 去操作这些元素。这就意味着 SVG 是高效且易理解的。

在第 1 章里，我们将夯实 SVG DOM 的基础知识，因为后面需要借助这些基础知识来实现复杂的 SVG 动画。我们将讨论一些基本的 SVG 语法，同时你可以依照自己的想法尝试操作并调试它们。我们不会深挖 SVG DOM 的每一个细节，因为这并不在本书的讨论范畴内。如果你想了解更多，推荐去看同样由 O'Reilly 出版的图书：

- *SVG Essentials*（作者 J. David Eisenberg）
- *SVG Colors, Patterns, and Gradients*（作者 Amelia Bellamy-Royds、Kurt Cagle）

这些都是与 SVG 相关的很不错的学习资源。

2 SVG DOM 语法

参考图 1-1，对应的代码如下所示：

```
<svg x="0px" y="0px" width="450px" height="100px" viewBox="0 0 450 100">
    <rect x="10" y="5" fill="white" stroke="black" width="90" height="90"/>
    <circle fill="white" stroke="black" cx="170" cy="50" r="45"/>
    <polygon fill="white" stroke="black" points="279,5 294,35 328,40 303,62
    309,94 279,79 248,94 254,62 230,39 263,35"/>
    <line fill="none" stroke="black" x1="410" y1="95" x2="440" y2="6"/>
    <line fill="none" stroke="black" x1="360" y1="6" x2="360" y2="95"/>
</svg>
```

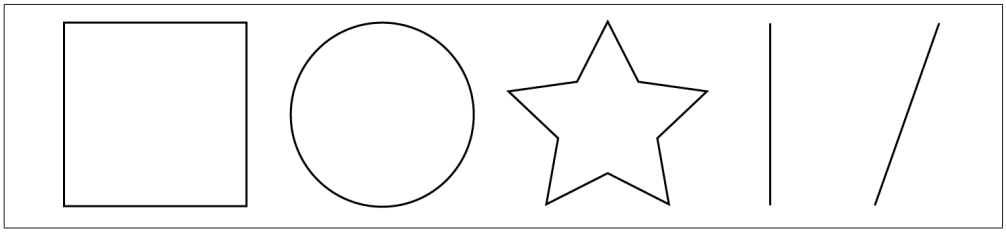


图1-1：SVG 编码的结果

让我们看看这个 SVG 的结构，大部分标签对于你来说都很熟悉吧？由于和 HTML 相通，SVG 的语法也是简单易读的。在根节点 `<svg>` 元素中，我们看到了 `x` 和 `y` 两个属性的声明值皆为 `0`，`(0,0)` 点是 SVG 坐标系统的起始点。同时 `width` 和 `height` 都有所指定，并且可以看到它们和 `viewbox` 的最后两个参数相同。

viewBox 和 preserveAspectRatio

SVG 的 `viewBox` 是一个非常强大的属性，因为是它真正地允许 SVG 画布无限延伸，并同时控制和精确定义 SVG 的可视空间。按照 `x`、`y`、`width` 和 `height` 的顺序，`viewBox` 有 4 个参数需要设置。可以发现，`viewBox` 的值并没有带单位，这是因为 SVG 可视空间并不是基于像素来设定的，而是一个可任意延展的空间，这样就可以适应许多不同的尺寸。为了便于理解，我们想象图 1-1 中的图形是画在一张方格纸中的（参见图 1-2）。

3 我们可以基于这张方格纸定义一个坐标系，这张方格纸本身是自我独立的。可以随意改变这张方格纸的 `width`、`height` 或者任意事物。例如，如果我们将 `<svg>` 的 `width` 和 `height` 减少一半，但是保留同样的 `viewBox`，那么结果就会是图 1-3 所展示的这样。

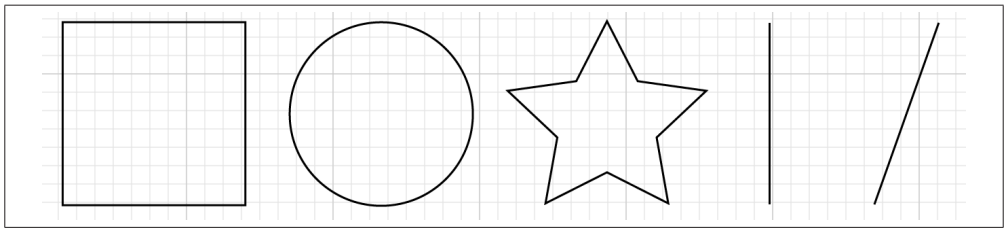


图1-2：SVG viewBox

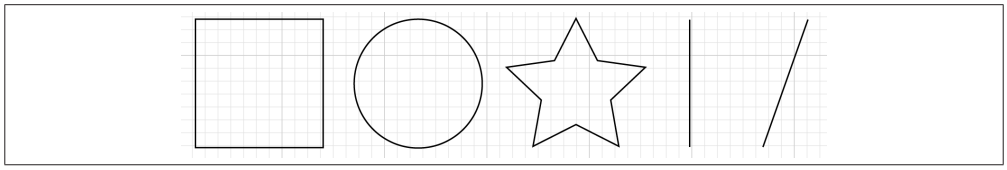


图1-3：viewBox 改变后的结果

译者注

我们可以理解为，所有图形都基于 viewBox 的坐标系，并绘制在 viewBox 中，而 viewBox 会自动根据 <svg> 的宽高和下面要介绍的 preserveAspectRatio 来进行适配。

这就是为什么 SVG 能成为响应式开发适配利器的其中一个原因。SVG 同时也会存储超出 viewBox 的信息。如果我们把一个图形移动到 viewBox 之外，情况如图 1-4 所示。

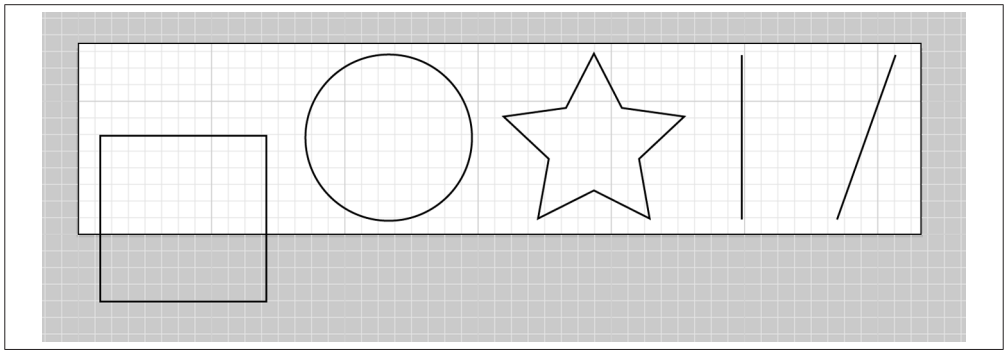


图 1-4：将图形溢出viewBox 空间之外的结果

如图 1-4 所示，只有白色的区域是用户能看到的，但实际上无论可视或是不可视区域内的绘图信息，都会被 SVG 元素保存起来。这个特性让 SVG 可被随意伸缩和裁切。对于响应式应用来说这是十分方便的，特别是用于 SVG Sprite 技术。

4

在这个例子中还隐藏了一个需要了解的关于 viewBox 的知识，那就是 SVG 的一个属

性——preserveAspectRatio。平时我们看到的许多 SVG 中都没有这个属性的声明，所以大多数人都不了解它。它的初始值为 xMidYMid meet，这使得 viewBox 以均匀的比例来适应 SVG 容器。

关于这个属性还有其他一些选项，其中第一个参数 xMidYMid 决定了 SVG 内的画布是否以均匀的比例来缩放以及缩放的位置，写法是以驼峰命名法构成的（注意 Y 是大写）。虽然初始的缩放位置默认从正中间开始，但也有一些其他对齐选项，例如 xMinYMax。 你也可以将其设置为 none，在这种情况下，SVG 的横纵比将会被忽略，整个 viewBox 画布会以压扁或拉伸的方式填充到 SVG 的可用空间中。

第二个参数可选的值只有 slice 或 meet。meet 会按照 SVG 的横纵比把整个 viewBox 的内容全部填充到 SVG 中，这个功能可以类比 CSS3 中的 background-size:contain;，图片内容会永远保持在容器边缘内部。

slice 允许 viewBox 超出用户的可视区，并在指定方向上填充可用区域。你可以将其类比 CSS3 中的 background-size:cover;，图片内容会超出容器边缘进行填充。

译者注

Meet 算法——计算 viewBox 的横纵比 width/height。

- 如果大于 1：取 SVG 宽为 viewBox 的宽，根据比例计算 viewBox 的高，然后根据 preserveAspectRatio 中的第一个参数调整 viewBox 的位置。
- 如果小于 1：取 SVG 高为 viewBox 的高，根据比例计算 viewBox 的宽，然后根据 preserveAspectRatio 中的第一个参数调整 viewBox 的位置。

Slice 算法：首先假设 SVG 的宽高固定。当选择 slice 后计算 viewBox 的横纵比 width/height。

- 如果大于 1: 取 *y* 轴方向为基准。整个 viewBox 的高设为 SVG 的高，然后再根据比例求出 viewBox 的宽，最后根据 preserveAspectRatio 中的 x[Min|Mid|Max] 调整绘制区域位置。此时 Y[Min|Mid|Max] 无效。
- 如果小于 1: 取 *x* 轴方向为基准。整个 viewBox 的宽设为 SVG 的宽，然后再根据比例求出绘制区域的高，最后根据 preserveAspectRatio 中的 Y[Min|Mid|Max] 调整绘制区域位置。此时 x[Min|Mid|Max] 无效。

只要 preserveAspectRatio 的第一个参数不为 none，就能保证 viewBox 是等比缩放的。

关于 PreserveAspectRatio 的更多知识可阅读张鑫旭的这篇文章，*http://www.zhangxinxu.com/wordpress/2014/08/svg-viewport-viewbox-preserveaspectratio/*。

更多资源

Sara Soueidan 做了一个极其直观和具有帮助性的 demo 让你边操作边理解 `viewBox` 系统（*https://sarasoueidan.com/demos/interactive-svg-coordinate-system/index.html*）。

Amelia Bellamy-Royds 在 CSS-Tricks 网站上也做了很多例子帮助你弄懂 `viewBox`（*https://css-tricks.com/scale-svg/*）。

Joni Trythall 也有很多很棒的关于 `viewBox` 和 `viewport` 的资源等你来体验（*http://jonibologna.com/svg-viewbox-and-viewport*）。

绘制图形

在上文绘制的 SVG 中，我们定义了 5 种不同的形状。`<rect>` 是指矩形或正方形，其中的 `x` 和 `y` 属性，相对于 `<svg>` 元素进行定位，在这个例子中是相对 `<svg>` 的左上角进行定位的，同时图形的 `width` 和 `height` 也使用同样的坐标模型：

```
<rect x="10" y="5" fill="white" stroke="black" width="90" height="90"/>
```

`fill`（填充色）和 `stroke`（描边色）属性被设定为 `white` 和 `black`。如果没有设定 `fill` 和 `stroke`，那么它们的默认值是 `fill: black;` 和 `stroke: none;`（没有描边）。

`<circle>` 定义和你想象的一样，是一个圆：


```
<circle fill="white" stroke="black" cx="170" cy="50" r="45"/>
```

`cx`、`cy` 定义的是圆的圆心坐标，`r` 为圆的半径。你也可以用 `<ellipse>` 绘制一个椭圆形，唯一不同的是椭圆形比圆多两个参数：`rx`（短轴长）和 `ry`（横轴长）。

`<polygon>`（多边形）是通过以空格分隔的数组列表的 `points` 属性来定义各个点的坐标的：

```
<polygon fill="white" stroke="black" points="279,5 294,35 328,40 303,62 309,94
279,79 248,94 254,62 230,39 263,35"/>
```

和你想象的一样，例如对于 `279,5` 来说，以逗号分隔，前一个数为一个点的 *x* 坐标，后一个数为一个点的 *y* 坐标。多个点共同组成了这个多边形。

`<line>` 是相当简单的：

```
<line fill="none" stroke="black" x1="410" y1="95" x2="440" y2="6"/>
<line fill="none" stroke="black" x1="360" y1="6" x2="360" y2="95"/>
```

`x1` 和 `y1` 为线的起点值，`x2` 和 `y2` 为线的终点值，在这里设置了两条语法相近的线，你可以对比着判断哪条为直线，哪条为对角线。

响应式 SVG、组和绘制路径

现在让我们看图 1-5 和相应的代码：

```
<svg viewBox="0 0 218.8 87.1">
    <g fill="none" stroke="#000">
        <path d="M7.3 75L25.9 6.8s58.4-6.4 33.5 13-41.1 32.8-11.2 30.8h15.9v5.5s42.6
        18.8 0 20.6" />
        <path d="M133.1 58.2s12.7-69.2 24.4-47.5c0 0 4.1 8.6 9.5.9 0 0 5-10 10.4.9 0
        0 12.2 32.6 13.6 43 0 0 39.8 5.4 15.8 15.4-13.2 5.5-53.8
        13.1-77.4 5.9.1 0-51.9-15.4 3.7-18.6z" />
    </g>
</svg>
```

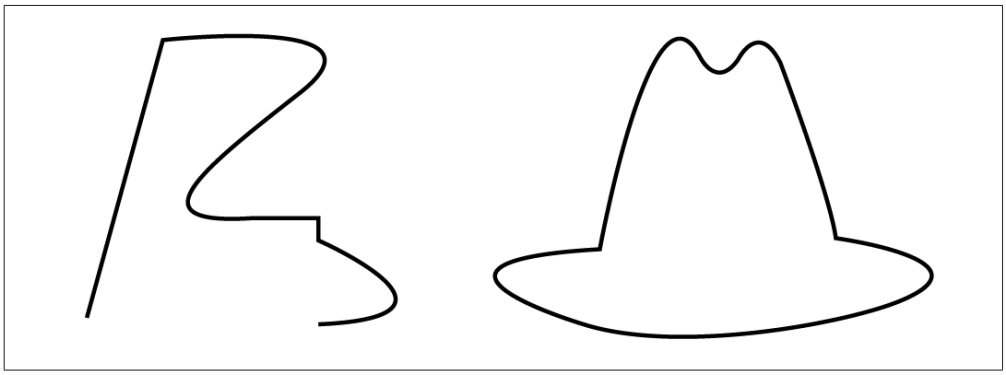


图 1-5：一个没有定义width和height的SVG

6 首先需要注意的一点是，这个 SVG 的 `width` 和 `height` 都没有定义，而实际上，我们可以将 SVG 的一些属性定义在其他地方（例如 CSS、引入 SVG 的 `<img>` 或 `<object>` 标签中）。

译者注

如 `<img src="test.svg" width="120px" height="120px">`

或

```
svg {
  width: 120px;
  height: 120px;
}
```

或

```
<object type="image/svg+xml" width="400px" height="400px" data="test.svg"></
object>
```

这种特性使得 SVG 具有很强的延展性，特别是对于响应式开发来说。

宽 / 高适配

使用 CSS 控制一个元素的尺寸，并且将尺寸都保存在一个 CSS 文件中，这种做法是较好且简单的。但是，我通常不会在 CSS 内定义 SVG 的 `width` 和 `height`，因为担心 CSS 文件不能被正常加载。如果同时也没有在 SVG 中设置 `width` 和 `height` 属性，那么 SVG 就会继承父级的宽高，这样就完美地达到响应适配的效果了。介于这个原因，当需要设定宽高的时候，可以在 SVG 中加上行列属性作为保障，因为 CSS 属性会覆盖行列属性的值。而如果 CSS 属性没有被正确加载，则会取用行列属性。

SVG 可依照百分比或者视窗单位进行适配，甚至还可以通过媒体查询来影响 SVG 的尺寸。而唯一不足的是，在这个例子中，你必须声明 `viewBox`，一个缺少 `width` 和 `height` 属性并且声明了 `viewBox` 的 SVG 元素的默认行为是继承包含自身的父级元素，这个父级元素可能是 `body`、`div` 或者其他元素。

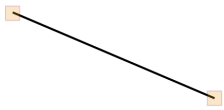
其次我想指出的是 `<g>` 标签，`g` 即是 group（组）。这个标签在 SVG DOM 中用于嵌套和集合多个元素。你也许注意到了，我们只在 `g` 标签中设置了 `fill` 和 `stroke`，而每一个 `path` 元素虽然没有设置，但是却应用到了 `g` 所设置的 `fill` 和 `stroke`。这说明在 `g` 标签中设置的样式可以影响它的子级元素。

最后一件非常重要的事就是解释 `path` 元素的语法，一条路径起始于 `d` 属性，`d` 即是 data，也就是一条路径的绘图数据。`d` 通常会以一个 `M` 或 `m`（即 `moveTo`，移动到）为第一个值，也就是确定一个新的起点。语法上和创建一个 polygon 或 polyline 是不同的，同时这个起点也不一定会在最终绘出的路径上。

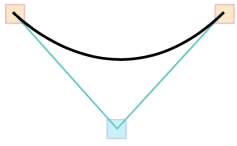
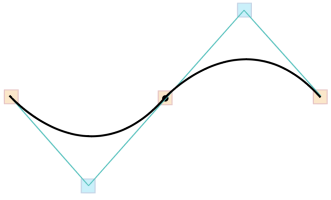
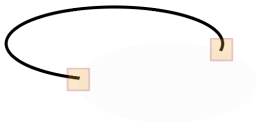
表 1-1 向我们展现了 `path` 中的各种命令的含义，命令都有大小写之分，大写命令定义的是相对路径，而小写命令定义的是绝对路径。

7

表 1-1　路径语法

字母命令	含义	图片展示
`M,m`	`moveTo`	路径起点
`L, l`	`lineTo`	
`H,h`	从当前位置绘制水平线	

续表

字母命令	含义	图片展示
V, v	从当前位置绘制垂直线	
Z, z	加到最近的 moveTo 命令的路径末尾	路径终点
曲线命令		
C,c	Cubic Bézier	
S,s	Reflecting cubic Bézier	
Q,q	Quadratic Bézier：双方共享相同的控制点	
T,t	控制点已经反应出来了	
A, a	椭圆的弧	

8

9 再回去看图 1-5 及其代码，可以清楚地发现 path 和 polygon/polyline 的区别在于使用了 z 命令作为路径的结束符。

进一步学习 path data 已经超出了本书的范畴，但是这里有 Sten Hougaard 制作的一个直观的 demo，可帮助读者更好地了解 path 语法（*http://codepen.io/netsi1964/pen/pJzWoz*）。

SVG 的导出、建议及优化

你可以完全手写 SVG，也可以使用 JS 库来绘制 SVG，例如 D3（*http://d3js.org/*）。但是，当你想设计和创建一个 SVG 时应该选用例如 Adobe Illustrator（下文简称 AI，见图 1-6）、Sketch 或者 Inkscape 这样的矢量图处理工具。通过使用这些工具，每一图层中的图片会以 g 标签的形式被导出，并将图层名赋值给 g 标签的 id。下面是一个导出 SVG 的示例，你可能会发现你的 SVG 多出了很多前面例子中没有的信息。

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Generator: Adobe Illustrator 18.1.1, SVG Export Plug-In . SVG Version:
    6.00 Build 0) -->
<svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
      width="218.8px" height="87.1px" viewBox="0 0 218.8 87.1"
  enable-background=
    "new 0 0 218.8 87.1" xml:space="preserve">
  <g>
    <path fill="#FFFFFF" stroke="#000000" stroke-miterlimit="10"
      d="M133.1,58.2c0,0,12.7-69.2,24.4-47.5c0,0,4.1,8.6,9.5,0.9
                c0,0,5-10,10.4,0.9c0,0,12.2,32.6,13.6,43c0,0,39.8,5.4,15.8,
      15.4c-13.2,5.5-53.8,13.1-77.4,5.9C129.5,76.8,77.5,61.4,133.1
      ,58.2z"/>
    <path fill="#FFFFFF" stroke="#000000" stroke-miterlimit="10"
    d="M6.7,61.4c0,0-3.3-55.2,20.8-54.8s-7.2,18.1,4.1,29.9
             s8.6-31.2,32.1-15.8S86.7,41,77.2,61.8C70.4,76.8,76.8,79,37.9,
    79c-0.4,0-0.9,0.1-1.3,0.1C9,81,40.1,58.7,40.1,58.7" />
  </g>
</svg>
```

这里贴出之前的示例便于进行比较：

```
<svg viewBox="0 0 218.8 87.1">
    <g fill="none" stroke="#000">
        <path d="M7.3 75L25.9 6.8s58.4-6.4 33.5 13-41.1 32.8-11.2 30.8h15.9v5.5s42.6
         18.8 0 20.6" />
        <path d="M133.1 58.2s12.7-69.2 24.4-47.5c0 0 4.1 8.6 9.5.9 0 0 5-10 10.4.9 0
         0 12.2 32.6 13.6 43 0 0 39.8 5.4 15.8 15.4-13.2 5.5-53.8 13.1-77
          .4 5.9.1 0-51.9-15.4 3.7-18.6z" />
    </g>
</svg>
```

10 你可以发现之前的示例更简洁，这是由于没有正确地进行优化操作，所以 SVG 代码很容易出现冗余。

AI 小技巧

当使用 AI 的时候，应该确保使用“文件→导出→ SVG（这个功能只支持 AI CC 2015.2 以上版本）”这样的步骤来导出 SVG，而不是直接选择“另存为”，然后就会弹出一个 SVG 选项对话框，如图 1-6 所示。

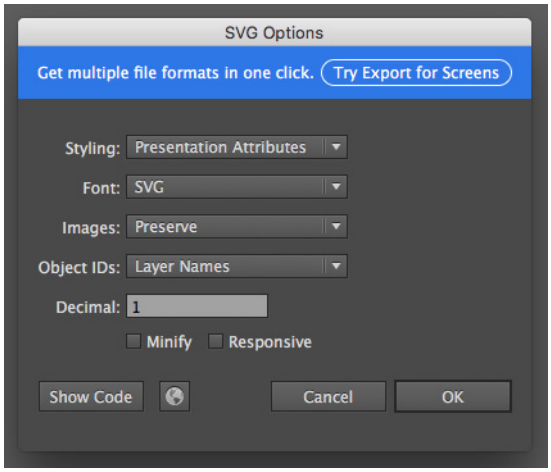


图1-6：从AI中导出SVG的设置

这种方式导出 SVG 会比直接通过“另存为”命令生成的没有进行优化的 SVG 在体积上更小，且绘图信息更精确。

我本人总会保留数份 *.ai* 的源文件，因为当需要大规模修改 SVG 的时候，在 AI 上不能很好地对 SVG 旧版本进行回溯。

AI 导出的 SVG 中有一些信息是十分有用的，但有些是可以丢弃的，例如 AI 导出的 SVG 中夹带的注释部分。版本信息和图层信息也不需要。当用于线上环境的时候，这些信息通常没有什么用，增加了体积还不便于传输，应尽量让 SVG 文件体积更小。

如果将 `x` 和 `y` 都设为 `0`（通常情况下），我们是可以把它们去掉的，唯一需要保留的情况是：当子级也有 SVG 嵌套需要定位的时候。

如果使用的是行内的 SVG，同样也可以去掉关于 XML 的定义。强调一下，在此书内自始至终都会使用行内的 SVG，这样能更好地控制 SVG 动画，减小出错的风险。但是有
11 些时候在动画中使用 SVG 作为背景图片也是非常不错的选择（我们会在第 3 章和第 4

章讲解 SVG Sprite 时进行更详细的讨论）。而当使用 SVG 做图片或背景的时候需要把 XML 标签声明加上，以兼容老式浏览器：

```
xmlns="http://www.w3.org/2000/svg"
```

如果你确定用不到，那么最好把它去掉。

也可以优化 path，AI 导出的 SVG 中的 path 数据会夹带许多浮点小数，通常情况下这些数是可以被四舍五入的。同时导出的 g 标签也会扰乱你的代码，让我们来看看下面几个可能需要优化的地方。

减少路径点

如果你想创建一个手绘图的效果，可以通过“对象→路径→简化”命令来帮你获取每个点的值。图 1-7 所示的是一张“简化”对话框的截图。你需要勾选 Preview 复选框来实时查看改变后的结果，因为在这个阶段，没调好参数可能会改变图片的质量。如图 1-7 所示，随着曲线精度的增高，图片质量会快速降低。这个例子中使用的曲线精度为 95%。随着精度的降低，会减少路径点数据，同时 SVG 文件的体积也会快速减小。

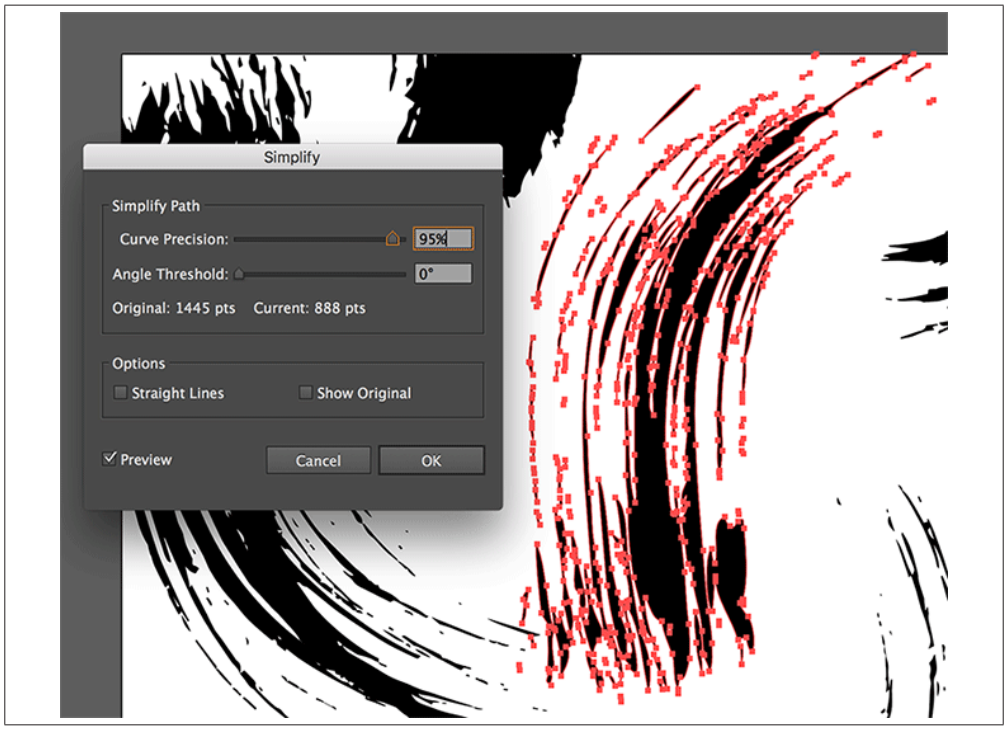


图1-7：AI的“简化”对话框，可以快速减小SVG文件的体积

12 上文提到的方法也许是完成简化最快的一种方式。而对于那些较小、不是太重要、不太复杂的 SVG 图层来说，我会选择用钢笔工具手动重绘它们。采用这种方式的效果不一定很显著，视图形形状而定。也许你付出了很多却收效甚微。

这听起来好像很难，不过你确实可以使用钢笔工具快速绘制出很多复杂的区域。同时使用路径选择工具可以将多个复杂的区域进行合并（参见图 1-8）。如果这看起来好像有点不对，不要怕，你可以略微减少合并后的图层的透明度（这样就可以看到你试图用简单图形来模仿下面用路径绘制的复杂图形了）。然后选择直接选择工具（工具栏中的白色箭头是快捷键），使用这个工具拖动图形点可不断优化你绘制的图层，直到效果和自动产出的结果差不多为止。这样处理的效果，细节上放大看起来也差不多，但是文件体积减小了。

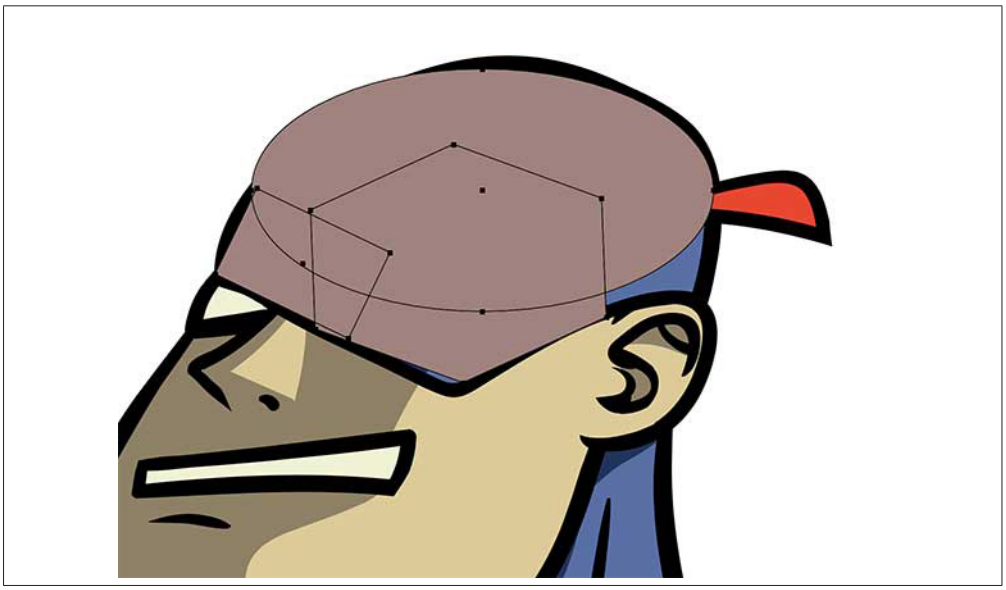


图 1-8：快速绘制多个图形，并将它们融合到一起，从而替代一个由SVG自动产生的路径图

优化工具

其实没必要手动去掉那些不必要的信息，因为开源社区提供了很多用于优化 SVG 的工具，它们可以帮助你减少代码体积。这些工具具有如下一些功能：自动四舍五入过长的数字，合并路径数据，移除非必要的图层等。

下面介绍一些有用的开源工具。使用这些工具，你可以直接看到优化后产出的代码和效果，这样有助于你选择合适的工具。

SVGOMG（*https://jakearchibald.github.io/svgomg/*）

这是由 Jake Archibald 开发的一款很棒的在线 GUI 优化工具，它底层是基于终端 SVGO 内核的。这个工具对于工作来说最稳健、最简单，其包含了很多优化选项。SVGOMG 还可以直接展示优化后的效果图，以及优化前后文件大小的对比。

SVG Editor（*http://petercollingridge.appspot.com/svg-editor-*）

Peter Collingridge 开发的 SVG Editor 工具与 SVGOMG 类似，不过功能略微少了一些。有一个不错的功能是，当你想轻调某一处细节时，可以在左边的输入框中进行修改，并且在右边能同时看到效果。 这个工具也是在线工具，具有不错的可视化界面。

SVGO（*https://github.com/svg/svgo-*）

SVGO 是一款基于 Node.js 的 SVG 优化工具，它并不是一个可视化工具，但是 SVG 组织提供了 SVGO-GUI 这款可视化工具。使用 SVGO 之前可能需要进行一些配置（安装 Node 下载依赖包等），不过如果相对于总是要切换浏览器界面来说，终端操作更加方便，那么选用终端化 SVGO 也是一种工作享受。这个工具和 SVGOMG 具有相同的功能。

如何决定改变或者调整某些优化选项呢？这依赖于你想实现一个怎样的动画。请合理优化你想要的结果，不要直接使用默认优化设置，这样能帮你节省大量的时间。在开发过程中，你可能发现有一段比较复杂的动画需要反复进行优化，这时我建议你在开发的同时打开代码编辑器、图片编辑工具以及优化工具，这样能尽可能地保持工作流无缝对接。

默认导出设置

当你导出 SVG 图的时候，需要注意一些默认的导出配置。

这里有一些我经常勾选或取消的默认配置。

- 清除 ID：这个选项会去掉你仔细为各个图层设置的名字。
- 去掉无用的 g 标签：当需要为多个图层同时制作动画的时候，可以用 g 标签将它们包起来，否则还是保持代码原样较好。
- 合并路径：这个选项多数时候都是工作正常的，不过有些时候，当合并多个路径的时候，你会发现不能通过 DOM 独立移动元素了。
- 保持代码缩进：只有当需要在编辑器上编辑 SVG 代码的时候才需要保持代码缩进，其他情况下请压缩你的 SVG 代码。

15 第 2 章

使用CSS制作SVG动画

大概因为 SVG 中也存在一个 DOM 的概念——类似标准的 HTML 语法的原因，你也许会觉得 SVG 看起来十分熟悉。实际上 SVG 也能通过 CSS 进行动画制作，这无疑是一项格外有价值的特性，因为对于前端开发者来说，使用 CSS 操作标签是格外轻松的。

让我们先对 CSS 动画做一下简短的复习吧，创建 CSS 动画的第一步就是先定义两个属性。首先，让我们来看看这个动画的关键帧（keyframe）：

```
@keyframes animation-name-you-pick {
    0% {
        background: blue;
        transform: translateX(0);
    }
    50% {
        background: purple;
        transform: translateX(50px);
    }
    100% {
        background: red;
        transform: translateX(100px);
    }
}
```

关于 keyframe 语法的小贴士

你可以使用 `from` 和 `to` 来代替百分比的定义，如果没有声明初始 keyframe 或者结束 keyframe，那么 CSS 会使用元素默认声明的属性。如果需要兼容很多浏览器，那么需要反复检查你的动画帧是否漏了初始帧和结束帧，否则可能在一些浏览器中会出问题，虽然这可能是由于一些浏览器本身的 bug 或者标准不统一所导致的。

当定义了这些 keyframe 的值后，有两种方式可对动画进行配置，这里有一个语法冗余的 16
版本，它将关于动画的每一部分的定义都单独列出来进行声明：

```
.ball {
    animation-name: animation-name-you-pick;
    animation-duration: 2s;
    animation-delay: 2s;
    animation-iteration-count: 3;
    animation-direction: alternate;
    animation-timing-function: ease-in-out;
    animation-fill-mode: forwards;
}
```

当然这里还有一个简短版的（这是我最喜欢用的，因为代码比较简短）：

```
.ball {
    animation: animation-name-you-pick 2s 2s 3 alternate ease-in-out forwards;
}
```

`animation` 的属性是以空格分隔的，除了一些关于数字的值以外，其他有关动画属性的声明顺序是可以互换的。只能用数字值来定义的属性需要遵照这样的顺序：`duration`、`delay` 和 `iteration count`（分别代表动画的总时间、延长几秒开始和执行的次数）。

应用了上面定义的动画帧的 `.ball` div 的结构也很简单（参见图 2-1）：

```
.ball {
    border-radius: 50%;
    width: 50px;
    height: 50px;
    margin: 20px;
    background: black;
}
```

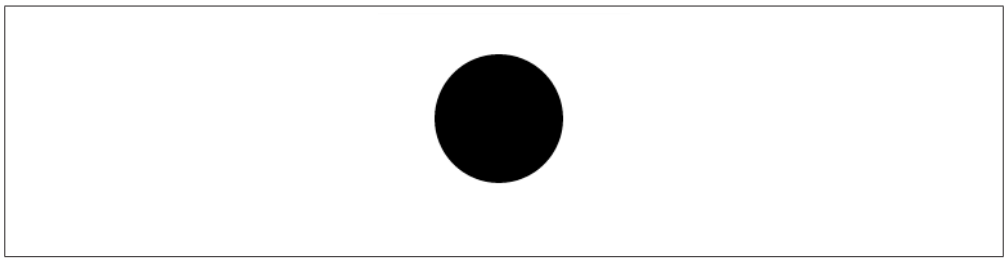


图2-1：应用`.ball` div的小球

我们通过上面的代码得到了图 2-2 所示的结果。

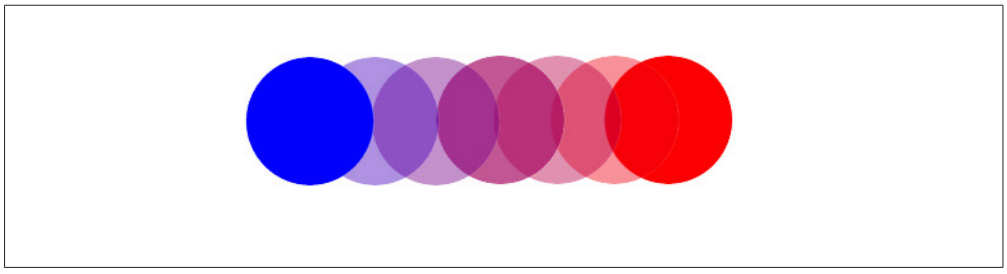


图2-2：CSS动画用于使用div创建的小球的结果

17 你可以在 *http://bit.ly/2lSB8KZ* 看到有关我创建的这个例子的完整代码。

如果你想了解有关 CSS3 动画属性更多的内容，例如什么是 `animation-fill-mode` 或者 CSS3 中有哪些缓动函数，又或是有哪些 CSS 属性适用于 CSS3 动画，请参考同样由 O'Reilly 出版社出版的、由 Estelle Weyl 编写的 *Transitions and Animations in CSS* 这本书。

同时你也可以参考由 Dudley Storey 编写的，由 Apress 出版社出版的 *Pro CSS3 Animation* 一书。

用 SVG 做动画

让我们来看看用 SVG 代替刚刚使用的 CSS 和 HTML 来声明的小球的效果，运用上一章学过的知识，用 SVG 来画一个和图 2-1 所示一样的小球：

```
<svg width="70px" height="70px" viewBox="0 0 70 70">
    <circle fill="black" cx="45" cy="45" r="25"/>
</svg>
```

我们定义了一个半径为 25px 的小球，并且把小球的中心移动到 SVG 画布的 `(45,45)` 点上。画布的宽高是 70×70px，所以也像刚刚 CSS 中声明的那样预留了 20px 的左上外边距。当然，我们同样也可以对 `<svg>` 运用 `margin` 属性来移动它，但与使用 SVG 本身提供的坐标声明比较来说，如果我们移动了一下这个小球，整个 `<svg>` 移动所占用的空间是宽加上 `margin`。

现在，如果把这个 `<svg>` 元素加上同样的 `ball` 类，那么就会运行相同的动画函数声明，然后我们就能看到如图 2-3 所示的效果。

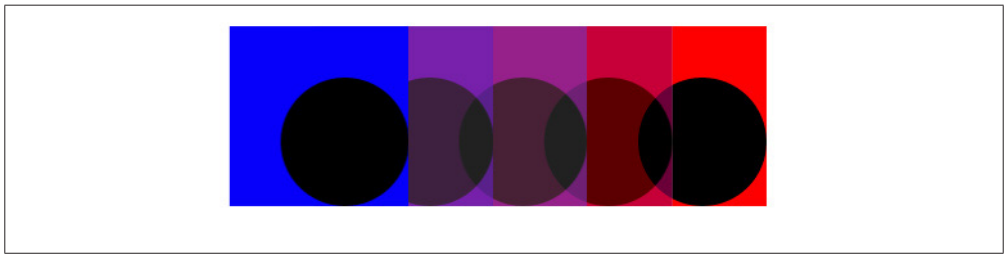


图2-3：给这个SVG加上 .ball 的效果

到底发生了什么？虽然小球像我们预期的那样仍然在移动，但是变色的背景却填充了整个 SVG，即整个 viewbox，这显然不是我们想要的。那么如果把 .ball 加给 <circle> 标签，会发生怎样的结果呢？让我们看看图 2-4。 18

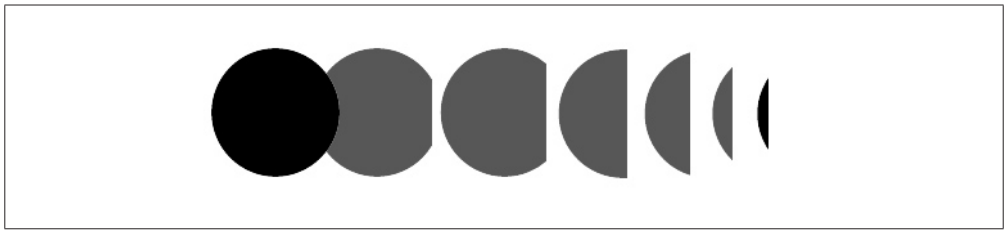


图2-4：将.ball加给<circle>的结果

根据这个结果，我们也大概能猜到是什么原因了：

1. <circle> 会在 viewBox 内运动，记住，我们前面讨论过，viewBox 就像是一个窗口，如果移动了 SVG 内部元素的属性，就好像在一个窗子里看外面的物体运动。超出窗口的元素会被裁掉，所以如果把 <circle> 元素移出了窗口的范围，效果就如图 2.4 所示的那样，被裁切或遮盖。

2. SVG DOM 就像 HTML DOM，但是它们略有不同。不允许对 SVG 元素使用 background 属性，在 SVG 中我们需要用到的是 fill 和 stroke 属性。对于 SVG 来说，外联样式表会覆盖 <circle> 的内联属性样式，所以统一将 SVG 的样式（fill 属性）写在外联样式表中。

把我们的 SVG DOM 稍微修改一下：

```
<svg width="200px" height="70px" viewBox="0 0 200 70">
    <circle class="ball3" cx="45" cy="45" r="25"/>
</svg>
```

对应的 CSS 代码如下所示。

19
```
.ball3 {
    animation: second-animation 2s 2s 3 alternate ease-in-out forwards;
}

@keyframes second-animation {
    0% {
        fill: blue;  // 注意，把 background 换成了 fill
        transform: translateX(0);
    }
    50% {
        fill: purple;
        transform: translateX(50px);
    }
    100% {
        fill: red;
        transform: translateX(100px);
    }
}
```

使用 SVG 代替 HTML 绘制小球的效果如图 2-5 所示。

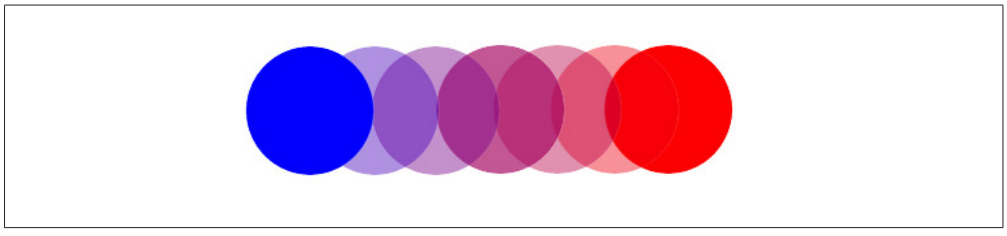


图 2-5：采用fill属性的效果，并且把 fill 写进外联样式表中

使用 SVG 绘图的优势

在需要制作由 CSS 样式控制的 HTML 动画的时候，为什么需要学习 SVG 呢?

首先，从一个小角度来看，例如要绘制一个简单的圆，我们需要写 4 行 SVG 和 CSS 代码，明显少于使用 CSS 和 HTML 来实现的代码量。SVG 的绘制是不同于 CSS 的，SVG 是以一种表象格式绘制的，让我们看看第 1 章所讲的绘制星星的代码：

```
<polygon fill="white" stroke="black" points="279,5 294,35 328,40
   303,62 309,94 279,79 248,94 254,62 230,39 263,35 "/>
```

使用如此少量的代码就可以绘制一颗星，这对于使用 HTML+CSS 实现来说是非常困难

的。即使使用预处理器来处理 CSS，编译出来的代码也不会这么少。

图 2-6 所示的是我用 AI 画的一个小例子。

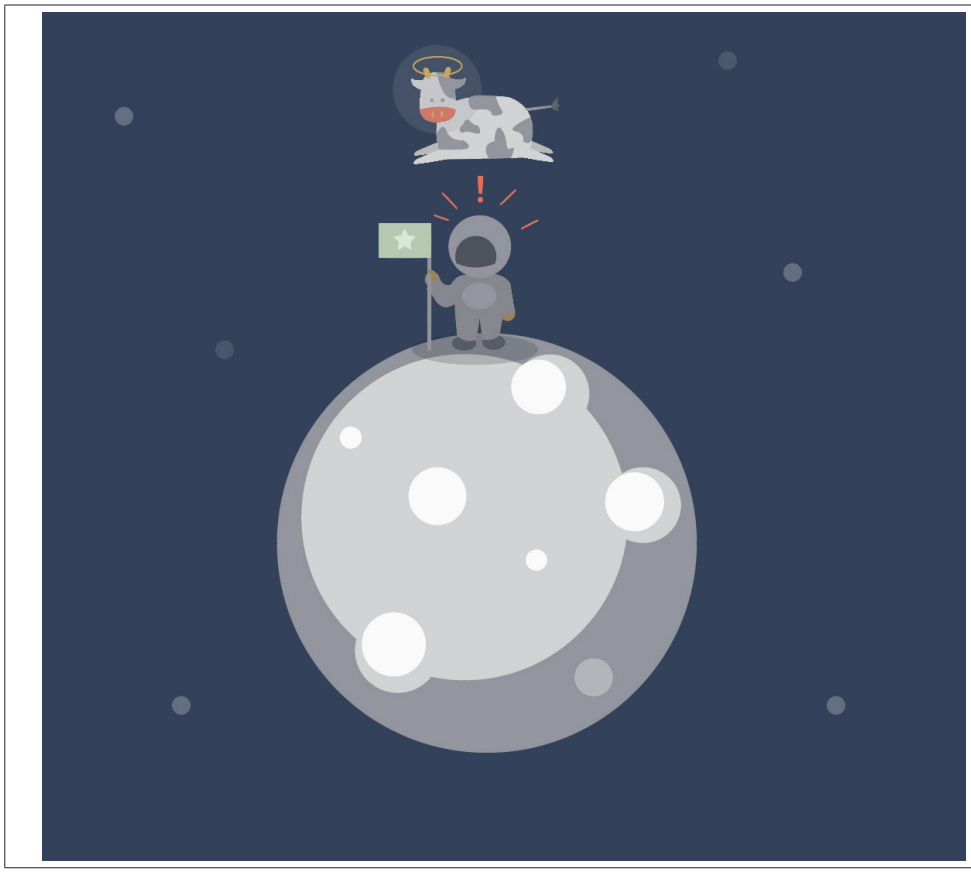


图 2-6：一个简单的插图例子

对于图 2-6，也可以使用 CSS 进行绘制。但是，如果我们和一名设计师共同开发一个项目，通常不会选择使用 CSS 来画。因为如果使用 CSS 绘制，再加上动画后逻辑会很复杂。如果使用 SVG，相对于 CSS 来说，绘制一张插画较为简单，而且更容易缩放，添加的动画也是响应式的，不用兼容各种不同宽高的屏幕。 20

整个插画的数据信息压缩后仅有 2KB，但是这样就能填充到整个屏幕的大小。相对于使用不同尺寸的位图进行兼容的做法，这是一个很激动人心的优点。

让我们用前面刚刚学会的 `<circle>` 来看看这些图形能够帮我们做些什么吧！我们可以使用 `<g>` 将奶牛的图层部分包裹起来，并让其跳跃到月球上，也可以让宇航员突然消失

和出现。甚至可以调整头盔图层的位置使其上下移动，给人感觉宇航员在寻找某些东西，这其实也是这个动画的最终效果。

21 顺畅的动画体验

对于 CSS 属性布局，我们可以使用 `margin`、`top`、`left` 等属性，表面上它们都能达到相同的预期效果——这虽然是很吸引人的地方，但是实际上浏览器对于不同属性的值并不是同等对待的，例如，如果使用 `margin` 就会涉及浏览器重排和重绘的问题。为了让动画简洁流畅，动画方面最好的解决方案是使用 `transforms` 属性来改变图形的形变，同时配合 `opacity` 属性来解决图形的淡入淡出。在标准动画中使用这些属性所达到的效果也是十分令人惊奇的。

在本书中，在我演示各种技术的时候会尽可能地使用上面介绍的这两个属性。同时需要特别注意的一点是，当前 SVG DOM 只在某些浏览器中存在硬件加速功能，例如，只有 Firefox 支持 SVG 硬件加速，而 Chrome 不支持，但是，即使浏览器对 SVG 存在硬件加速等优化，你仍然应该用 `transforms` 进行 SVG DOM 的移动，千万别用 `margin` 或者其他 CSS 位移属性（涉及浏览器的重构和重排）。

在本书出版的时候，IE 浏览器和 Edge 还不能完全支持使用 `transforms` 操控 SVG 元素，但是你可以在 Windows 开发者反馈调查网站上对一些有关的提议进行投票。

那么在这些浏览器支持使用 `transforms` 操控 SVG 之前，你最好还是用 SVG 原生的属性（这比较痛苦，因为要用 JavaScript 做一些兼容）或者使用 GreenSock 动画库提供的 API。GreenSock 是专门用于制作 SVG 动画的工具库，它在内部做了对浏览器的兼容，并兼容到了 IE 9。

如果想了解“如何保持布局绘制成本较低”的相关知识（一些与 Chrome 相关的学习资源），可以去看 Jank Free 写的 *High Performance Animations* 一文。

如果想了解一些特别属性在绘制方面的资源消耗的相关知识，可以去 CSS Triggers（*http://csstriggers.com/*）上查询。

第 3 章 23

CSS 动画和手绘SVG Sprite

SVG 作为一个图标格式来说是非常合适的，不过，下面我们将更进一步地了解，如何通过三种不同的技术使用 SVG Sprite 执行复杂的动画。前两种技术和 cel 动画非常类似，第三种技术我们将在第 4 章详细阐述，这种技术我推荐在更复杂的响应式动画和交互性的 SVG 中使用。

从设计的角度来看，这是一种更先进的动画技术。我们现在在这本书中探讨该技术，因为实际的动画可以使用纯 CSS 实现。本书会以一个渐进式的方式讲解动画技术（首先是 CSS，然后 JavaScript 库，最后是原生的 JS），不过你可以选取适合你的章节阅读，第 7 章提供了以上动画技术的比较。

使用 steps() 和 SVG Sprite 制作关键帧动画

如果你看过 *Looney Tunes* 或者老版迪士尼的动画，可能会对其流畅的动作有很深的印象，因为它的每一帧动画都是手绘的。这样的效果其实可以使用 SVG Sprite 在 Web 上展现。我们可以站在以前动画绘制者的肩膀上去使用新的开发技术。

在所有基于 Web 的动画技术中，步间（step）动画是最容易使用以前手绘的 cel 动画的。cel 是 celluloid 的缩写。它是一种透明材质，动画绘制者用它在前面样片的基础上绘制，从而定义了一个序列并且创造出运动的错觉。该技术有点类似于以前快速翻阅的小人书。每张图纸以一帧一帧的形式被放置到电影中。为了节省时间，这些图示通常会包含多 24
层——你可能也并不想一次一次地为了展现同样的场景去重复画背景吧。

为了节省画图的步骤，会先画出背景，然后是人物，有时甚至是人物的脸部，比如嘴巴或者眼睛，这些后面有变化时只需要调整一下即可。图 3-1 和图 3-2 所示的就是一个分层的例子。

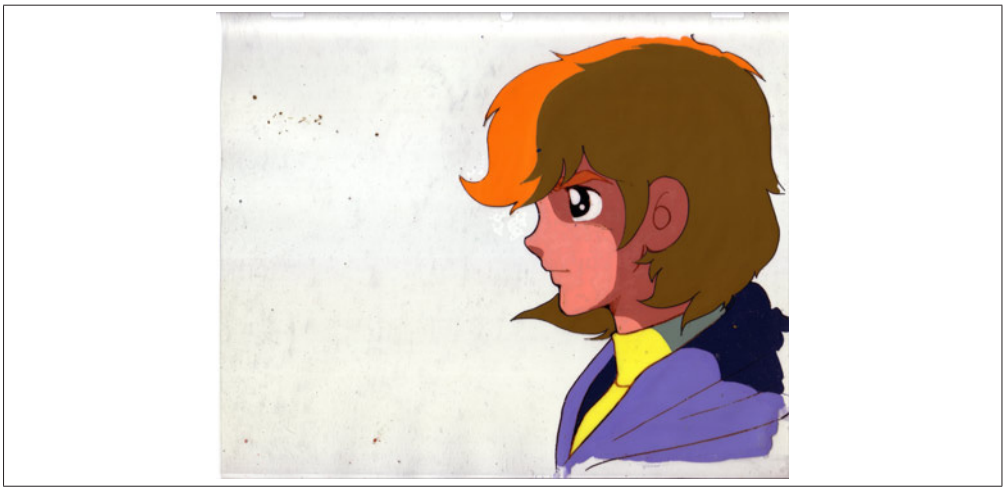


图3-1：手绘的透明cel（由John Gunn提供）

cel 动画的相关知识

你可以认为该技术就像写一个 Web 页面的模板：从基础的模板开始，然后创建更小的模板。所以，你可以控制一些独立于其他片段中发生的事物。

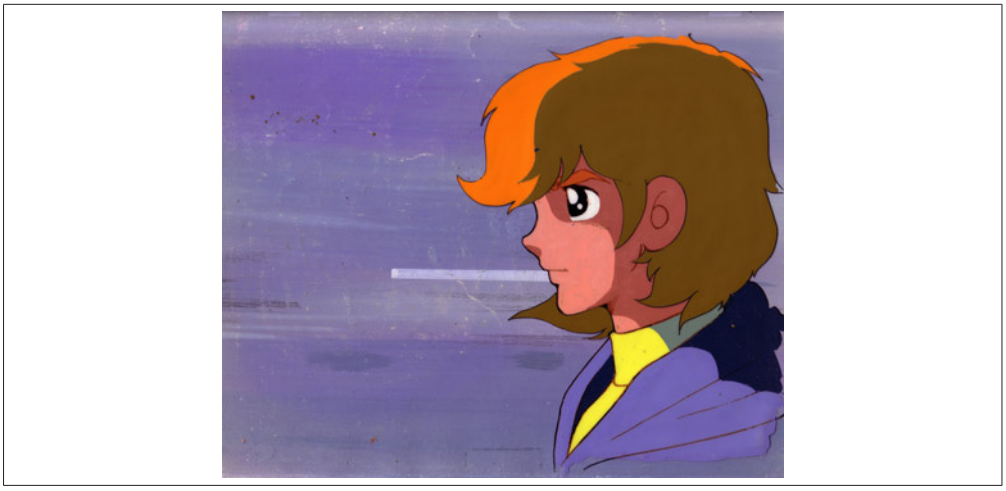


图3-2：手绘有背景的cel（由John Gunn提供）

25 我们可以使用单个静止的背景模拟这个过程，然后快速地在顶层展示一系列图片。这可以在没有真实插值的情况下，呈现出运动的错觉。为了替代这些独立的图片，我们将同时减少 HTTP 请求的数量并且使用单个 SVG Sprite 图来简化关键帧（参见图 3-3）。该技

术对比简单的变形动画而言，更适用于复杂的形状和富有变化的运动。

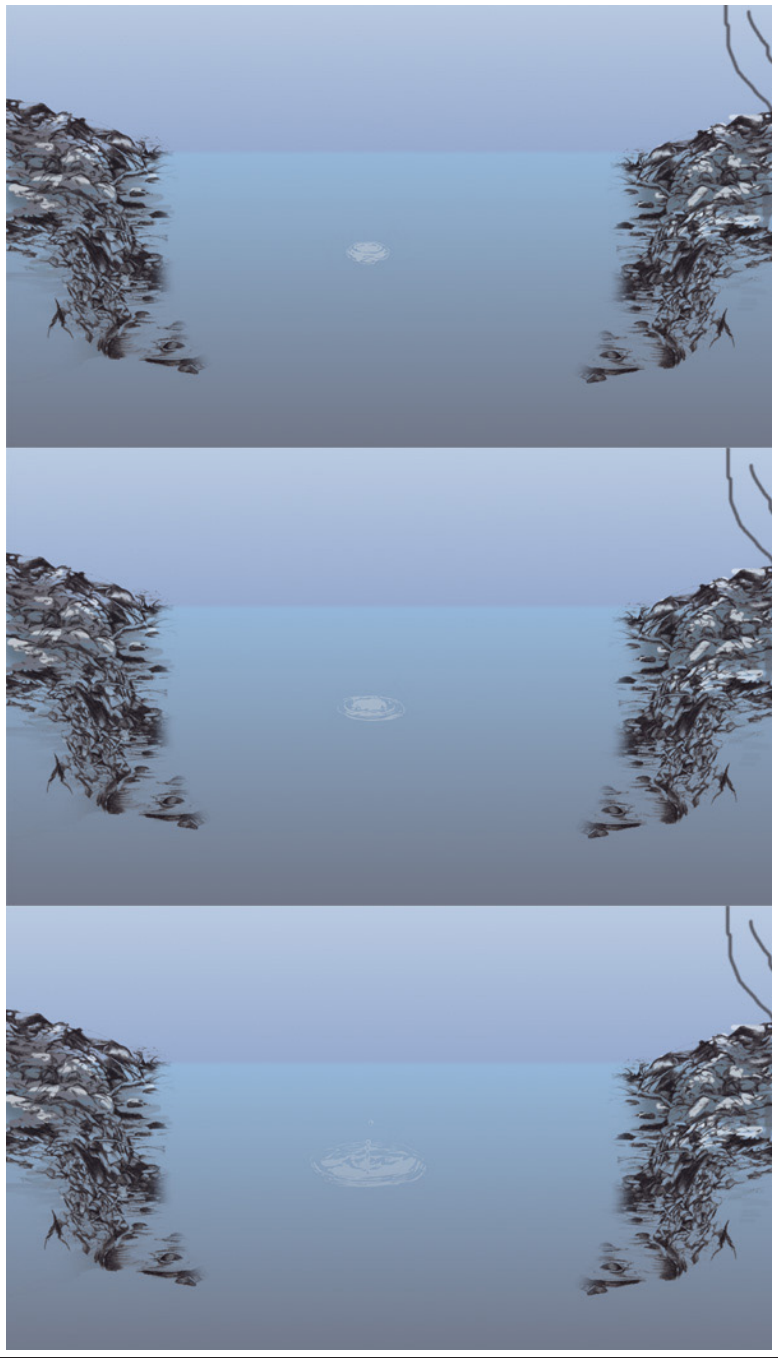


图3-3：水飞溅的动画

因为该技术严重依赖于设计，所以，我们将先熟悉设计方面的工作流程，然后再看具体的实现代码。你可以在我的 CodePen 上（*http://codepen.io/sdras/pen/LEzdea/*），找到最终的动画效果。

通常，在 Web 上展现插值（例如，快速切换）图片时，为了让动画展示流畅，我们需要使用最大的帧速率(fps)。而该技术则是这个规则的一个例外。因为我们需要绘制每一帧，所以需要尽可能地减少手绘量（参见图 3-4）。以前，动画绘制者花费大量的时间在实际运动和最少的图纸量之间寻找一个平衡点。老电影是以 24fps 进行拍摄的，而动画绘制者基本上使用“双倍拍摄”（意味着 2 帧一张图纸，或者说帧速率为 12fps）作为运动错觉的标准。在低于上述情况时，人的眼睛会察觉到轻微的卡顿（大多数动画制作者认为这是一个很有创意的决定）。对于 1.8s 的动画，需要 21 张图纸。21 代表着我们选的帧的数量，不过，你可以自己决定有多少帧。

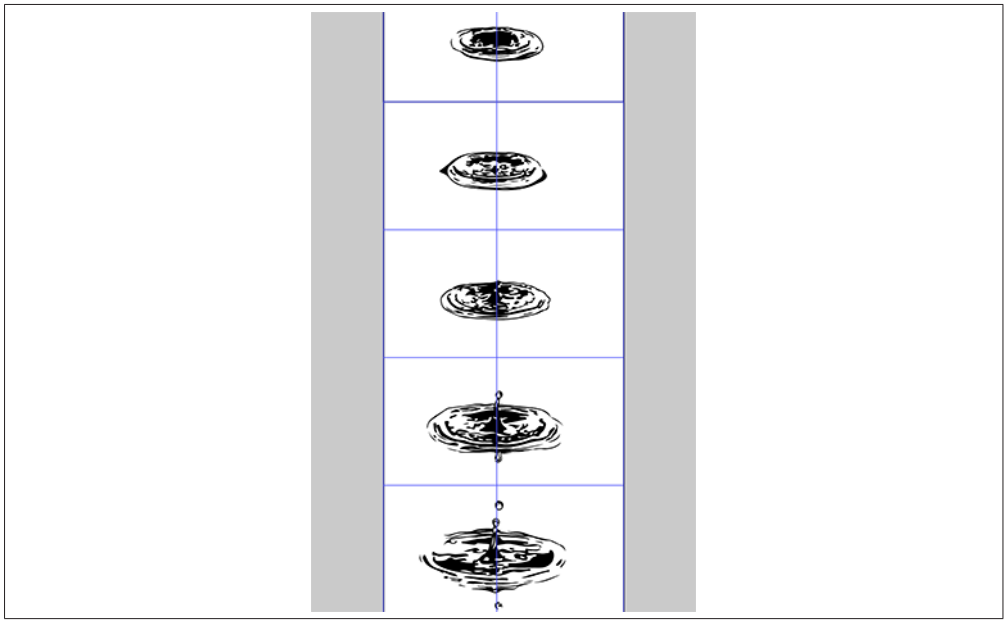


图3-4：在画板上绘制每一帧

27 在 Illustrator 中使用模板绘制

这里有两种方式可以针对 Sprite 动画创建一组图纸；这两种方式大致相同，不过它们对于图像使用不同的自动化流程。在两种工作流程中，我们遇到了一个挑战，即，如何让图纸连续地放在帧的中心，从而形成一个大的 Sprite 图。如果图纸错位了，当播放每一帧时，就算是完美的图纸看起来也会有瑕疵。

对于这项技术，我们一般使用 Illustrator，不过你可以使用 Sketch 或者其他图形编辑器。首先，我们需要知道动画的帧数，然后在一个方向上乘以 21，这决定了我们画板的长度。我们在画布中拖出一个区域，选择 Object → Path → Split Into Grid 命令。然后输入想要的行数（如果希望使用一个水平的 Sprite 图表就输入列数）。最后，选择 View → Guides → Make Guides 命令，随后我们的模板就生成好了。

如果你是直接在图形编辑器里绘图的话，我建议你将第一张图纸放在第一个区域里，然后在它的周围再创建一个区域，并且让帧都处在导轨里。你可以使用对齐线，或者按住 Shift 键拖动，将任意内容复制到下一个区域里（包括区域中的帧），这样可以让其连续分布。接着，重复使用区域中的帧到下一个导轨处。

你可以使用直接选项工具（白色的箭头）拖曳和重塑每一帧图像的片段。注意，不要一开始就在这里复制粘贴你所有的工作，当你参考前一帧去创建后一帧时，上面的流程才能有很好的效果。

28

你可以截取一些屏幕快照，然后通过一些定格图，将每张图像放置在 Illustrator 文档中并进行追踪。你既可以通过 Illustrator 自带的追踪工具，也可用钢笔工具来试一试手绘的感觉和更精确的路径。

在这项流程最后，将会得到一个很长的 Sprite 图表。我们能够直接将它导出为 SVG 或者 PNG。另外，我们将使用 PNG 和一个全局的类样式，在 Modernizr 下，实现一个渐进增强的方案（对于 Modernizr 的知识可以参考本章后面介绍的“使用 Modernizr”的内容）：

```
.splash {
    background: url('splash-sprite2.svg');
    ...
    animation: splashit 1.8s steps(21) infinite;
}

/* fallback */
.no-svg .splash {
    background: url('splash-sprite2.png');
}
```

这里，尽管降级方案可能没有太大必要，不过，建议你查阅一下 *caniuse.com* 对于 SVG 的支持。

在 SVG 编辑器或图纸中逐帧绘制并且使用 Gruntiocon 生成 Sprite

第一步仍然是用 Illustrator 绘图，不过，你可能想要手绘的感觉。如果是这样，那么就手绘出图纸，然后把画好的图纸扫描一份。在以前的老画室里，一般使用的是灯箱和赛璐珞片，这样可以方便一步一步追踪以前的图纸。不过，现在并不需要这些材料去实践该技术。通过将一个台灯放在一个玻璃桌面下，就可以轻易制作出一个简易的灯箱。这个装置可以发射足够的光线，让你查看这些相等规格的不透明的副本图纸。如果你想创建一个新帧，可以将一张白纸或者皮纸放在上一张图纸上，然后仔细绘制，直到有一系列的帧。你可以继续扫描这一套图纸并将其矢量化，注意要正确地放置，减少不透明度和导轨量。

如果你更喜欢在编辑器中绘制每一帧，但是又不知道总共要画多少，那么可以单独绘制每一帧，然后每次稍微移动图纸，再将每个新的版本保存到文件夹中。Illustrator 新的导出设置已经很完善了，你可以选择去掉旧草稿。注意，要通过 Export → SVG 命令进行导出，而不是使用 Save As → SVG 命令。你需要弄清楚一开始保存的是 SVG 而不是 AI 类型或者其他类型。接着，你可以使用 Grunticon（*http:/www.grunticon.com*）自动压缩和生成 Sprite 图表。在 CSS-Tricks 上，有一篇很好地阐述了如何使用 Grunticon 的文章（*http://css-tricks.com/inline-svg-grunticon-fallback/*），可以去看看。

29

就我个人而言，我觉得如果你手绘图纸，那么最重要的是确保每一张画纸对齐，然后使用 Grunticon。Illustrator 的模板技术可以让你一眼看到所有的图纸，这可以让你对正在做的动画有一个全局的了解。

用简易代码模拟复杂运动

这种 Sprite 图可以使用较少量的代码来生成大部分预期的动画。我们尽可能地让代码 DRY（Don't Repeat Yourself 的首字母缩写）、简单、干净。这种运动最关键的点在于我们主要依赖于 Sprite 图，而不是使用很多的代码去实现运动的感觉。

我们使用绝对位置定位了一块较小的运动区域，因为我们想在桌面端和手机端得到一致的效果。我们的预期是循环整个图像，但是能够单独地停在每一张独立的图片上。幸运的是，在 CSS 中可以使用 `steps()` 来完成。我们已经在设计阶段做了大量的工作，所以，模拟效果的代码不会很复杂。

这里不需要复杂的百分比和关键帧。我们所要做的就是使用图像的高度，然后在 100% 的关键帧里，将其负值赋给 `background-position` 属性：

```
@keyframes splashit {
    100% { background-position: 0 -3046px; }
}
```

这里，我们并不需要使用 .container-fluid，因为它非常容易使 SVG 在移动端上占据整个屏幕。在 splash div 中，我们使用 steps() 算出 SVG 中帧的数量，然后模拟动画：

```
.splash {
    background: url('splash-sprite2.svg');
    ...
    animation: splashit 1.8s steps(21) infinite;
}
```

使用 SVG 而不是 PNG 能够让我们在屏幕上展示更清晰的图片，不过，它也需要简单地提供一个渐进增强。我们使用 Modernizr 在 body 上创建一个钩子，然后可以使用 PNG 来模拟动画：

```
/* fallback */
.no-svg .splash {
    background: url('splash-sprite2.png');
}
```

30

我们一般不会轻易使用 PNG，因为在不同分辨率下，它有可能会变得模糊，而 SVG 会保持原有的清晰度。

使用 Modernizr

Modernizr（*http://modernizr.com/*）是一个特性检测库。它可以让你使用 Web 上的高级特性，它是一个高度自定义的库，会在 body 元素上写入你可以用来 hook 的相关的类，比如前面例子中的 .no-svg 标签。我强烈建议针对你特有的需求，使用自定义编译版本，因为整个库有点大，而你只使用其中的一小部分。

简单重复行走

如果把 steps() 从上一个动画中移除，你会发现一些有趣的事情。现在，不再是创建一个流畅的移动绘图，而是背景开始滚动。我们可以利用这一点，用空间布局和运动去做一个完善的分层背景。

我们现在着手做一个简单的重复行走动画（*http://codepen.io/sdras/pen/azEBEZ*）。大致布景为一个幽灵形状的人形走过一个循环的、多维的、粗略的背景。

我们可以使用前面介绍过的 cels/steps 技术，结合用来显示重复行走的图纸来创建上述动画。我们将使用一个手动动画技术，通过移动颜色不一样的帧来改变颜色。另外，我们可以使用带有色调旋转的滤镜去改变颜色，但只要我们是通过手工来创建这些帧的话，改变颜色所需要的工作量就很小了。滤镜在性能上的损耗，虽然不是很大，不过我们没必要使用它。

31

CSS Filter 案例

如果你打算使用滤镜，有很多网站介绍了 CSS 滤镜效果。例如：

- HTML5 Demos（*http://bit.ly/2lSARYv*），上述图像的来源。
- Bennett Feely 写的 CSS 滤镜效果集（*http://bennettfeely.com/filters/*）。
- CSSReflex（*http://www.cssreflex.com/css-generators/filter/*）。
- Una Kravets 写的 CSSGram（*http://una.im/CSSgram/*），其混合了几个滤镜，有点像 Instagram 的效果。这也是我最喜欢的一个网站。

注意，动态滤镜效果非常影响性能。我尽可能不做滤镜动画，不使用 `setTimeout`。在我需要它的时候，我会添加指定的属性或者 CSS，然后移除它。

`steps()` 和 `animation-duration` 的比应该在 `12fps` 的范围内，这点依然很重要。我们可以通过改变 SVG Sprite 图表的 `background-position` 属性来使图像滚动起来。为了确保一致性，我们让所有的背景图的尺寸都是一样的（参考图 3-5）。如果你熟悉 Sass 的话，可以使用 `@extend` 进行设置：

```
/--extend--/
.area {
    width: 600px;
    height: 348px;
}
```

32

```
.fore, .mid, .bk, .container { @extend .area; }
```

33

为了创建一个让人印象深刻的循环流畅的动画，我们需要在 *x* 轴上将这三个背景图无缝进行衔接，以至于当它们滚动的时候不出现缝隙。这可以通过让每张图底部紧贴，或者，在这种情况下，使用一个足够稀疏的图像，使其能够完整透过，如图 3-6 所示。如果你打算使用后面一种，那么当构建图形时，需要在一些图形编辑器中，比如 Illustrator 或者 Sketch 中，确保开始状态和结束状态看起来正常。

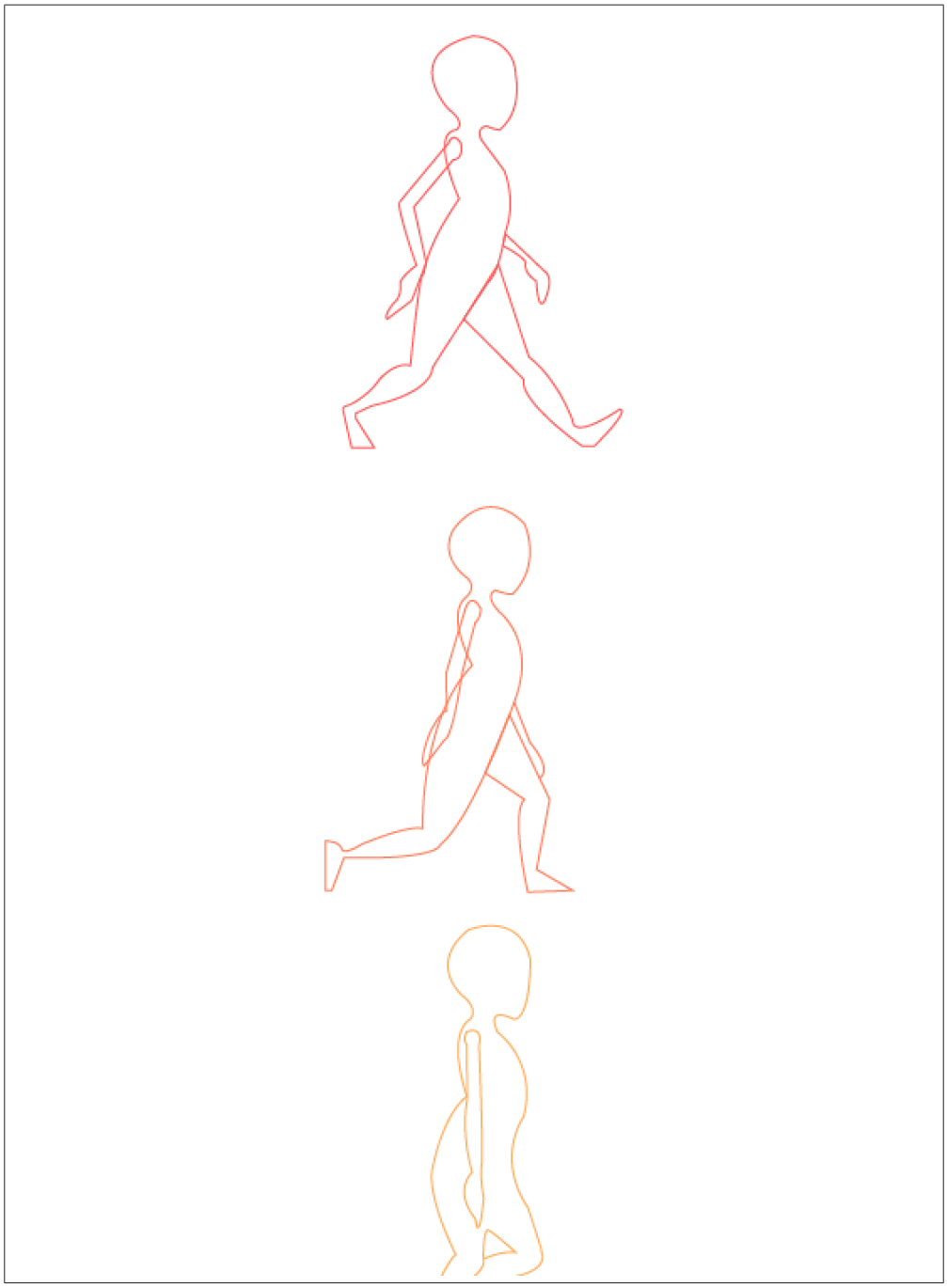


图3-5：图像的流动性和一致性

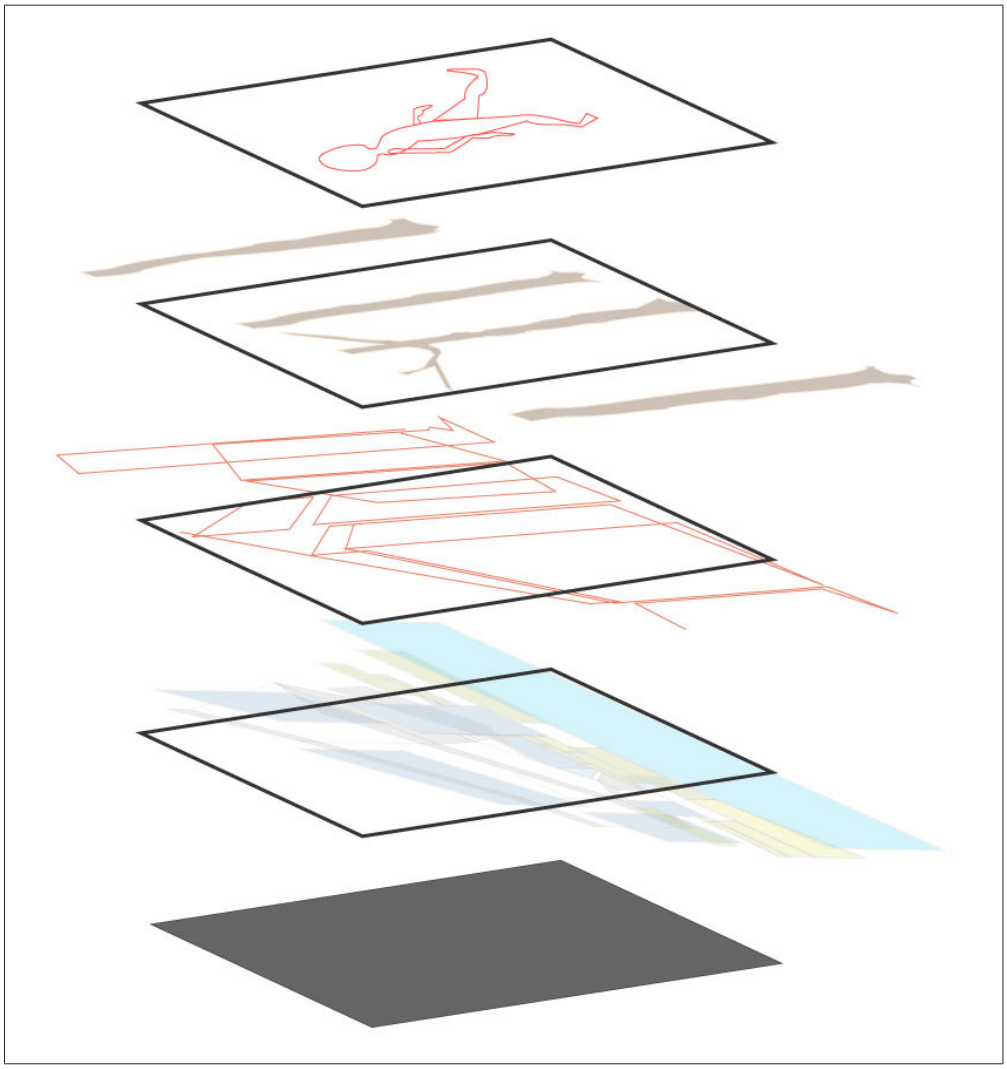


图3-6：我们将使用SVG创建一个深度错觉

34 每个元素都使用相同的关键帧动画值，但是我们会以它们 z-index 的减小来使每个元素动画时间一并减小。如果环顾一下四周，你会发现，离你近的物体会有更鲜明的对比，并且比离你更远的物体移动更快。所以，我们通过增加背景中 SVG 动画的第二个整数（从而拥有更长的动画），来使动画产生类似这种效果。这产生了一种很棒的视差效果。在这个例子中，有三个视差效果的背景图，但不包括人形：

```
.fore {
    background: url('fore.svg');
```

```
    animation: bk 7s -5s linear infinite;
}

.mid {
    background: url('mid.svg');
    animation: bk 15s -5s linear infinite;
}

.bk {
    background: url('bkwalk2.svg');
    animation: bk 20s -5s linear infinite;
}

@keyframes bk {
    100% { background-position: 200% 0; }
}
```

我们并不需要这种动画的多个间隔时间，因为关键帧已经为我们做好了插值。如果以后滚动 Sprite 图表的像素发生变化，我们也并不需要更新数据，因为已经将其值设置为百分比了。负值的延迟确保了动画能在开始的时候立即执行。所有的 SVG 都已经优化并且具有一个 PNG 的降级方案。

35

第 4 章

创建响应式 SVG Sprite

SVG 的可伸缩性可能是这种图片格式最强大的特性。使用 `viewBox` 属性和对图形路径的了解，可以动态地将 SVG 裁剪为任意大小。我们在坐标空间中所做的操作都会被保留。

如果将一个普通 SVG 上的 `width` 和 `height` 属性移除，将会看到一些有趣的事情。SVG 会填满视窗的 `width`，并且保持里面 DOM 的高宽比不变。

如果我们在 SVG 放大或缩小之前使用 CSS 关键帧或者 JavaScript 代码移动 SVG 元素，比如 `circle` 或者 `path`，那么这些变化同样会在图形中被缩放。这意味着如果通过百分数、CSS 的 flexbox 属性或者其他技术去缩放一个复杂的 SVG，那么你的动画也会被相应缩放。你不需要为了手机或者其他尺寸的设备调整任何代码，只需要关注如何一次写出正确的代码即可。

完整的动画应该能被完美地缩放。在下面的 CodePen 例子中（*http://codepen.io/sdras/full/jPLgQM/*），你可以在动画运行时任意地对其进行缩放，然后观察它的表现。这对于响应式的发展很有利。图 4-1 所示的动画使用了一个完整的流式办法。

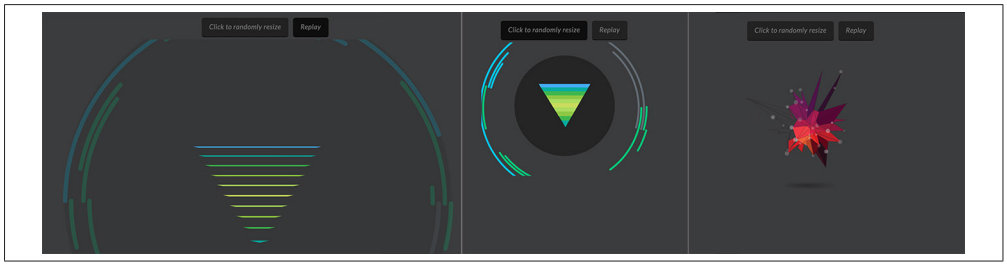


图4-1：不同状态的同一动画，其大小不同

36 我们首先将整体设计出来，然后慢慢进行展示。图 4-2 所示的是初始的 SVG（在添加任

意动画之前）的效果。

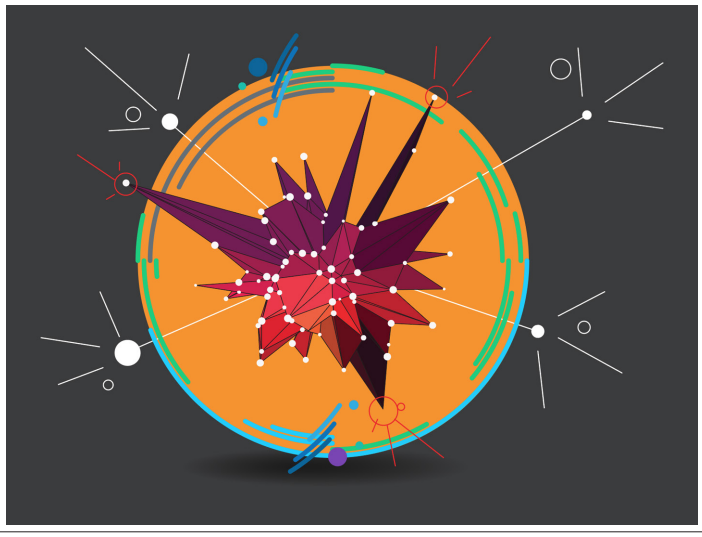


图4-2：Illustrator原始设计稿——首先设计所有的东西，然后慢慢展示需要的部分

我们有两种方法可以设计响应式 SVG。在本章中，我们将着重了解通过 SVG Sprite 图的方法，这和在第 3 章中介绍的类似，它很容易配合 CSS 的使用。在第 15 章，我们将讲解一些，当隐藏、显示、重叠、重排内容时，所使用的更高级的 JavaScript 方法。

用于响应式的 SVG Sprite 图和 CSS

37

Joe Harrison 在图 4-3 中阐述了一个非常好的办法。通过使用折叠的 SVG Sprite 图来减少移动端需要的视觉信息（*http://responsiveicons.co.uk/*）。当视窗从移动端变为桌面端时，我们将借鉴这项技术，并且创建一个相似的更复杂的 Sprite。

随着屏幕尺寸的缩放，图形可以灵活调整并缩小或者展示视觉上的复杂性。这对于用户来说很友好，因为他们不必在小小的屏幕上看到复杂的视觉图形，而这些信息往往都是让人烦躁的噪声。动画可以根据排版和布局做出相应更改，比如，适应视窗和简化设计。

我们将使用一个响应式的发光的字母（*http://codepen.io/sdras/full/xbyopy/*）来演示如何调整一个单独的插图（参见图 4-4）。该设计的灵感主要来自《凯尔斯之书》，这是一本让人难以置信的中世纪装饰手稿。该设计主要展现了一张独立的图纸是如何适应不同的视窗尺寸的。我们将以该设计开始，作为我们的“地图”。其他人可能有不同的想法，比如直接在浏览器中开始或者绘制草图。这都没关系，只要你选的方法能让你有所收获即可。

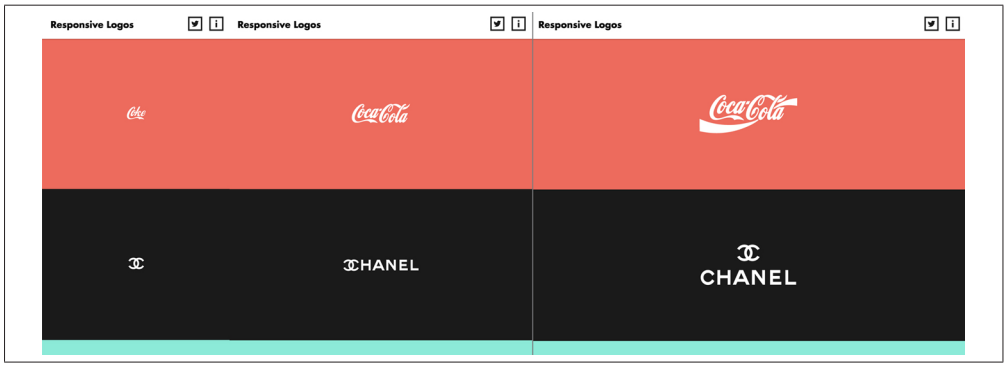


图4-3：Joe Harrison的经典SVG 版 Logo Sprite

图4-4：设计我们的地图

分组和导出

既然我们已经知道最后的结果是怎么呈现出来的，那么就可以和视窗的宽度关联起来，以分组的方式来重构设计。我们也可以对第一版和第二版中重复的图形进行简化，将共享的部分做成一份副本即可。

所有元素都被赋予语义化的 ID 名字，比如 mountain 或者 bridge，细小的图形同样可获得一个共享的 ID。如果第一张插图是 `kells1`，第二个插图的分组就为 `kells2`，第 3 个就是 `kells3`。

为了让 SVG 在共享容器值之间可伸缩，每一次插图的尺寸应该一样，剩下的则交给 SVG 内置的响应式处理来解决。

最后创建一个拥有两块相同宽度区域的 Sprite 图表，这样可以一次缩放整个图像（参见图 4-5）。第一张图片是最复杂的，它主要针对的是平板电脑和桌面电脑的用户。

图4-5：一旦减少重复，就可以准备导出

40 当得到了上述 Sprite 图后，就可以使用 SVGOMG 的 Web 模式的 GUI 去优化该图，检查是否有失真，清除不必要的 ID 的选项，合并无用的分组。然后，可以在需要的时候，
41 将 ID 改为类名，清除一些导出文件中没用的设计。我一般手动或者通过查找替换的方式完成这件事，不过完成这件事的方法有很多。

优化后的 SVG 是直接内嵌在 HTML 里的，而不是像之前的技术那样，使用背景图的 URL 属性。现在，我们将 SVG 隐藏，然后以移动端优先的方式进行展示：

```
@media screen and ( min-width: 701px ) {

    .kells3, .kells2 {
      display: none;
    }
}
```

为了重新定义每个版本的动画，可以参照 `viewport`，稍微调整一下动画的参数：

```
[class^="mountain"], [class^="grass"] {
    ...
    transform: skew(1.5deg);
}
@media screen and ( min-width: 500px ) {
    [class^="mountain"], [class^="grass"] {
        transform: skew(2deg);
    }
}
```

此时，为了让 SVG 更灵活，我们将 SVG 的 `width` 和 `height` 移除，并且设置 `preserveAspectRatio="xMidYMid meet"`（因为这是默认值，所以严格来讲并不需要）。通过这些更改，SVG 可以适应容器的大小。而我们将容器的大小设置为百分数形式（flexbox 或者其他响应的容器都可以在这里使用）：

```
.initial {
    width: 50%;
    float: left;
    margin: 0 7% 0 0;
}
```

viewBox 的技巧

这里有一个问题——即使我们在底层设置了一个类名，并且隐藏它，但是 `viewBox` 依然会计算那块区域，会存在一个空白的间隔。为了计算那块区域，可以改变 SVG 中的 `viewBox`，只展示顶层部分：

```
viewBox="0 0 490 474"
```

不过，这仅仅是为两个大的版本提供的，viewBox 给 SVG Sprite 图的部分提供了一个窗口，而最小的版本现在是模糊的。所以，我们需要调整一下。这和在 CSS 中更改背景位置去展示不同 Sprite 图的部分是类似的。不过，因为需要改变 SVG 的属性，所以要用到 JavaScript，仅使用纯 CSS 是做不到的：

42

```
var shape = document.getElementById("svg");

// 媒体查询事件处理器
if (matchMedia) {
    var mq = window.matchMedia("(min-width: 500px)");
    mq.addListener(WidthChange);
    WidthChange(mq);
}
// 媒体查询变化
function WidthChange(mq) {
    if (mq.matches) {
        shape.setAttribute("viewBox", "0 0 490 474");
    } else {
        shape.setAttribute("viewBox", "0 490 500 500");
    }
};
```

现在，在 W3C 的 GitHub 上有一个讨论（*https://github.com/w3c/fxtf-drafts/issues/7*），即是否将上述调整方式写入 CSS 规范；Jake Archibald 已经提了提议（*http://bit.ly/2mANBmP*）。如果提议被采纳，那么就可以使用媒体查询去更新所有的 viewBox，并且只需要使用一种语言即可。

当水平调整浏览器窗口的大小时，SVG 视窗会进行相应变化去展示想要展示的那一部分。现在，我们的代码已经写好了并且准备开始实施动画。

响应式动画

当从图形编辑器中导出 Sprite 图时，其中每一个元素都有独一无二的 ID。对于重复的元素，我更倾向于使用类名，所以通过查找和替换的方式将 ID 改为类名（Illustrator 依然会给每个类名添加一个唯一的数字，我们可以通过 CSS 中的属性选择器来选中它们）：

```
[class^="mountain"], [class^="grass"] {
    animation: slant 9s ease-in-out infinite both;
    transform: skew(2deg);
}
```

在下面代码中，给指定元素添加了变形动画，这可以让 keyframe 动画更完美和更精确。该动画假定 keyframe 中 0% 对应于元素的初始状态。为了创建一个高效的循环，需要在动画序列的中间部分设置动画的变化：

43
```
@keyframes slant {
    50% { transform: skew(-2deg); }
}
```

一些元素可以共享一个常见动画，比如点和星星。我们可以复用这些声明，只要根据需要调整延迟时间。将延迟时间设置为负值，这样可以在开始的时候就立即出现动画，即使这些元素动画是交叉的。当在 0% 和 100% 时，动画关键帧将会使用元素的默认样式（如果它们没有被定义）。我们将使用该特点尽可能地减少代码量：

```
@keyframes blink {
    50% { opacity: 0; }
}
[class^="star"] {
    animation: blink 2s ease-in-out infinite both;
}
[class^="dot"] {
    animation: blink 5s -3s ease-in-out infinite both;
}
```

我们还需要添加一个视窗的 `<meta>` 标签，这能让我们控制不同设备上的页面宽度和缩放。最常用的设置如下所示：

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

第 5 章 45

不使用任何额外库来创建 UI/UX动画

在前几章中，我们主要介绍了独立的 SVG 动画。在本章将介绍更常见的 UI 和 UX 元素，这些元素可以用 SVG 实现，并使用 CSS 驱动动画。之后，将专注于构建一个常见的带有可变换效果的图标，通过这个例子，你可以了解从开始到结束开发动画的整个过程，并将此过程集成到你自己的开发流程中去。

当用户浏览一个网站（或任何环境，或照片）时，他们正在尝试建立一个空间地图。在这个过程中，没有什么东西能像运动中的东西一样吸引他的注意力。

从生物学角度来说，人类是向感知运动的方向进化的：因为从进化的角度来说这是生存的需要。基于这个原因，一份制作良好的动画可以引导用户操作。动画可以帮助和强化我们在网站中构建的空间地图，并且带来一种更深刻地理解用户体验的感觉：获取信息后把信息放回原位，而不是放到什么突然出现的地方。

用户体验模式中的上下文切换

在讨论如何使用 SVG 动画构建典型的 UI/UX 交互之前，我们先来聊聊为什么要做动画。降低技术门槛是必要的，但是正确地使用动画同样重要。

你是否有过这样的经历，在一天的工作中别人总是打断你，让你做不同类型的事情？当你无法进入一种以流程为基础的工作方式时，工作会让你感到沮丧，同时会让你感觉混乱、无效率。用户使用网站时同样会有这种感受。 46

当访问一个网站时，你的大脑会使用一系列的快速眼动来创造空间关系。你从来没有真

正地“看着”一个图像：你的眼睛不停地移动来理解图片中的事物，从而创造一张图像的“心智地图”。有关眼动热图的示例，请参见图 5-1。

图5-1：用快速眼动来构建空间意识的过程中，所有的焦点都在网站的热点图上

创建一个网站时，其实我们是在为用户创建一个心智地图。对网站交互的改变会打破这种心智地图。Modal（模态对话框）是一个很好的例子：它们经常从一个地方冒出来，破坏用户的体验，也是我所称为的“野蛮用户体验”的一个例子。

47 有一种动画，可以通过顺应用户的心智地图减少上下文切换中的不顺畅：用户将从一致的区域中检索和访问信息，用户的体验会随着用户的需求而流动，整个体验会变得更加流畅。如何创建一个可引导用户的动画是需要一些思考的，所以让我们来分解一下可以做的事情：

- 变形
- 展现

- 隔离
- 样式
- 预期提示
- 交互
- 节约空间

在深入解决方案之前要注意，以上任何一个都有可能被“过度设计”。此外，人类的大脑已经进化到会特别注意一些移动。这种进化特征是为了保持安全和警觉；屏幕上出现意外移动时也会触发脑部肾上腺素的分泌。网站是一个静态的、没有动画的无聊网站；但是当涉及 UX 动效时，小而美是关键。

为了显示一个动画怎样保留用户的上下文的，我创建了一个例子（*http://codepen.io/sdras/full/qOdwdB/*），去看一下再继续往下读，之后还会用到这个例子。

变形

48

“变形”可能是帮助用户保留上下文最简单的方法。变形意味着相同的元素可以在不同的语境中传达多条信息，引导用户浏览而不会突然改变任何东西。考虑上面提到的动画，在 CodePen 示例中的这个交互元素中有多种变形形式。在这个例子中，一个元素经过一帧后变成了下一个：热点图标扩展为一个对话框，联系人按钮成为标题，文本框缩小为表示加载的点等，为用户提供了平滑的体验。

SVG 和 CSS 都是实现这些 UI 动画的好选择，而这两种方式均有自己的优缺点。通过使用 `border-radius`（*http://bit.ly/2lHGrjB*），CSS 可以平滑地在圆角与直角之间切换。它也可以优雅地处理大量的尺寸变换；而 SVG 会在图形切换时出现像素锯齿。但 SVG 是为绘图而构建的，它非常适合创建复杂的形状。

可以使用 JavaScript 和 GreenSock 的 MorphSVG 插件（*https://greensock.com/morphSVG*）来做路径甚至形状的补间动画，使用这种技术制作 SVG 动画是很好的：与 Snap.svg 甚至不太支持的 SMIL 不同，MorphSVG 可以轻松地在不同数量的路径数据之间进行转换，从而可以创建大量的精彩效果。如果有兴趣了解更多有关 SVG 变形的内容，请参阅本书第 10 章中介绍的内容，我们将在那里进行深入讨论。

展现

“展现”是保留用户上下文的一种非常简单的方法，但有些场景的展现会破坏用户上下文。下面我们以经典的模态对话框为例进行讲解。这是一个调用时出现 UX 的例子，但这并没有保留用户的上下文：模态对话框突然打破了用户的焦点和他们创建的空间映射。作为一个用户，即使模态对话框里包含了我需要的信息，有时候我也会关闭它，因为实在

是太碍眼了。

模态对话框本身并不是罪魁祸首，而是因为这通常是我们实现“展现”的方式。图 5-2 所示的是一个保留用户上下文的模态对话框例子：它从用户触发点开始，并替换其自身（*http://codepen.io/sdras/full/yOjWdO/*）。这两个过程之间有一个过渡，作为一个用户，我能知道这些信息在哪里“存活”，以便再次检索。

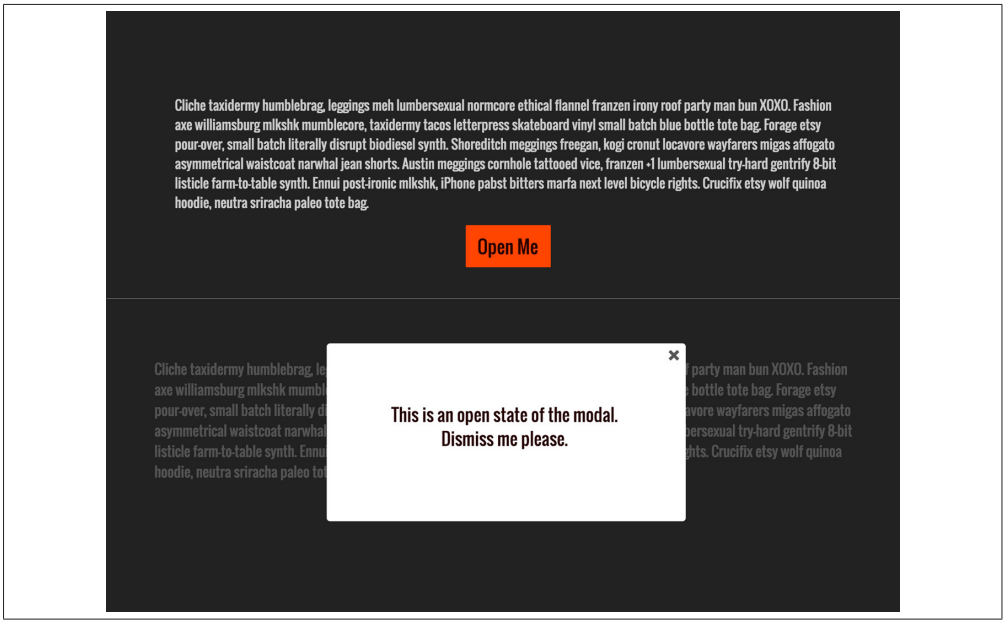


图5-2：模态对话框从其原点打开和关闭的状态

49 你也可以在之前的例子中看到这一点。地图上只标明了我当前的位置，当我触发相应控件时会看到更详细的信息。没有必要在页面中展示所有内容，当需要时能找到在哪里获取即可。

隔离

人们已经发现，在浏览网页的时候，我们不断地扫描页面来创建一个空间地图，隔离不同的区域可以帮助我们更快地了解信息。UI 可能变得凌乱：缩小选择减少了用户需要做的决定，这有助于用户提升对页面的支配度。

考虑图 5-3 所示的例子（*http://codepen.io/sdras/full/qOdWEP*）。起初，页面上有很多信息，使用户很难专注于一个区域。如果我们稍微调整一下 UI（这个例子中添加了 hover 状态），就可以将用户的注意力集中起来。

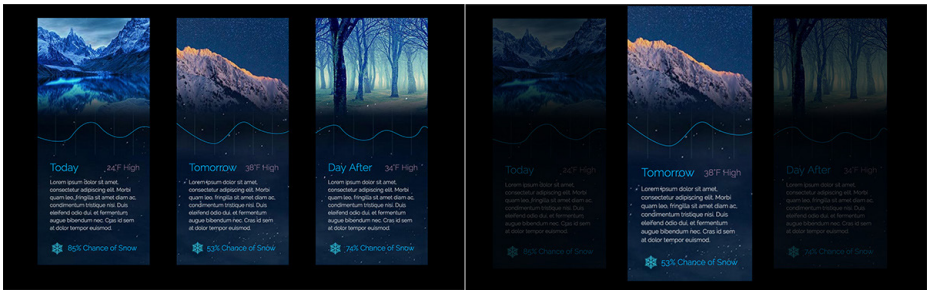


图5-3：通过隔离信息和遮盖其余的区域，用户能够更好地浏览和阅读网站上提供的信息

样式

50

样式、设计、品牌和缓动结合得非常紧密。当把自己的动画风格统一在品牌中（这是应该的）时，这便是“动效设计语言”。动效设计语言，对于让每个人理解页面上需要包含哪些动画类型是非常重要的。因此，可以通过在变量和交互中重用缓动的方式来保持代码整洁，并在整个站点甚至跨多个平台上保持一致的行为。我既不在 Android 上写 Java，也不在 iOS 上写 Swift，但是通过对这些应用程序的动画创建一个风格指南，我可以保持这些平台与网站的一致性。

如何让缓动发挥作用？缓动是动画品牌的主要组成部分。当你向银行或金融机构这样的公司提供服务时，其缓动方式更偏向 Sine 或 Circ 形式；当你在 MailChimp 或 Wufoo 这样更有趣的公司工作时，那么弹跳（Bounce）或伸缩（Elastic）的缓动形式就会更合适。（请参阅下页的“强调缓动”，以了解 Sine 与 Bounce 两种缓动方式的不同。）

以下是一些网站，可以选择更容易用于项目中的方式。

- CSS：*http://cubic-bezier.com* 和 *http://easings.net*
- GSAP：*http://greensock.com/ease-visualizer*
- React-Motion：*http://bit.ly/2mH7nvT*

51

强调缓动

缓动可以彻底改变动画的观感和强调点。Linear 和 Sine 缓动在数学上是线性的，且会在两个状态之间有一个过渡；而像 Bounce 或 Elastic 这类缓动方式，会在两个状态之间来回移动，从而产生一种潜在的更有趣的跳跃感。

在特定的操作或事件中，你可以使用缓动函数产生的效果来提醒用户，这种方式与设计者在调色板中使用着重色产生的效果是类似的，可参见图 5-4。有时访问网站时，你会注意到在多个地方使用一种主要颜色，而另一种着重色与此颜色形成鲜明对比。着重色用于提示用户进行单击按钮（CTA）等操作。这些 CTA 大部分是网站真正用来赚钱的操作，所以这些操作是否能从网页的其他部分夺人眼球是至关重要的。

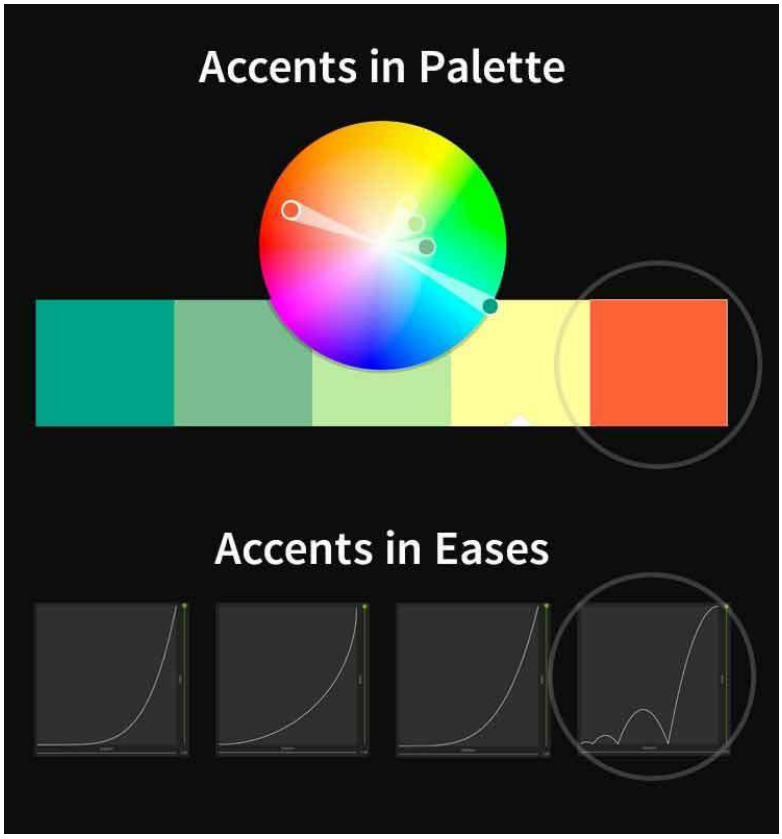


图5-4：正如我们在调色板中使用强调来引起注意一样，也可以利用缓动作为强调

我们可以使用之前介绍的缓动技术。在前面的例子（*http://codepen.io/sdras/full/qOdwdB/*）中，所有的缓动方法都是 Sine 方式的简化，它们更接近于平滑的线性。我们唯一一次使用 Bounce 这种缓动方式，是用于确认该表单已经提交完成并且成功了。

有关动画中的声音和音调的更多信息，请参阅 Val Head 的 *Designing Interface Animation*（Rosenfeld Media）。

预期提示

Eli Fitch 在 CSS Dev Conf（*http://bit.ly/2mGXAGq*）进行了一次名为“感知体验：唯一真正重要的事情”的演讲，这是我最喜欢的演讲之一。他讨论了如何通过像时间轴和网络请求的方式来衡量一件事情，因为它们更可量化——因此更容易测量——而衡量用户在访问网站时的感知体验是更重要的，非常值得花费时间和精力来研究。

麻省理工学院的 Richard Larson 在讲话中指出，“人类将被动等待高估了 36%”（*http://bit.ly/2lkBmJ0*）。这意味着如果你不使用动画来加速表单提交的体验，用户会感觉到它比开发工具记录的速度慢得多。

通过提交表单来向网站提供信息的用户，会经历一段时间的等待：他们不知道发生了什么，他们的信息被提交到哪里，或者信息是否有效。他们的信息通常需要不止一秒的时间来被处理，这使预期的行动非常重要。

预期状态的其他情况有可能是：

- 下拉式选择会改变页面的上下文
- 加载状态
- 按下一个按钮
- 拒绝登录
- 数据被保存

当发生这样的变化时，在页面上进行大量提示一般是没有意义的，但是仍然可以为表示页面的状态已经改变或正在进行改变创建一个上下文。考虑到我之前提到的技术，你可能会问：

- 这个过渡状态是否会吸引用户，还是仅仅是一个简单的到达最终状态的手段？
- 这种过渡状态是否可以重复用于其他情况？需要将其设计为具有足够的灵活性以适应多种位置和异常条件吗？
- 动画是否需要表达动作？一个例子是用户保存一些尚未完成的东西时，一个人性化的“等待”动作将有助于用户和网站的沟通。

为用户显示加载状态不仅可以告知他们后台发生了什么，如果它是一个自定义加载器，还可以让等待时间感觉不那么长，而且可让你的网站或应用程序展现出更高的性能。

图 5-5 中的效果用来展示高性能的动效带来的作用。

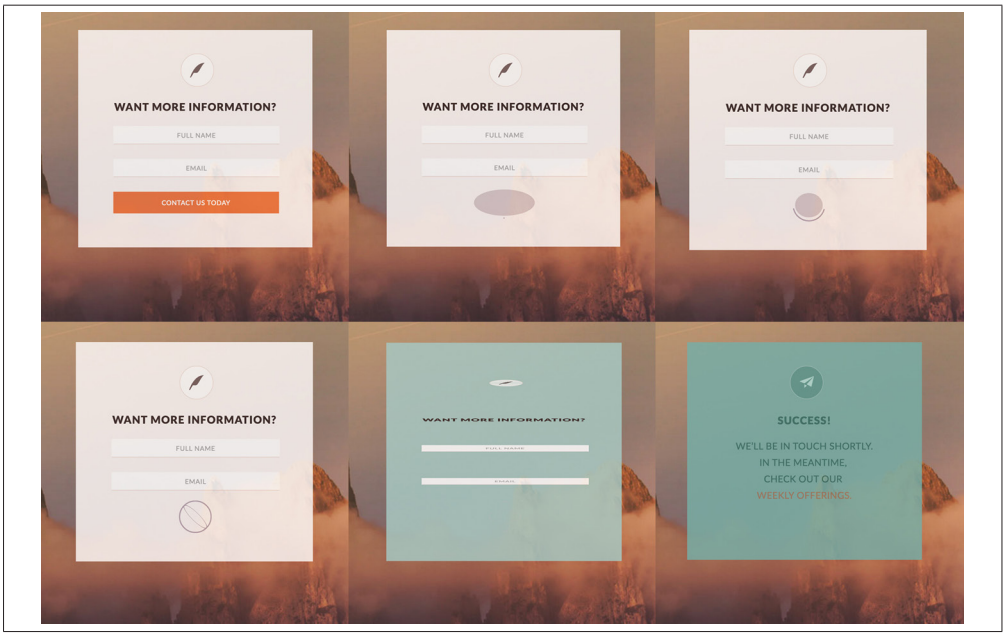


图5-5：显示变形加载状态和成功状态以减少所需等待时间

53 等待动作是直接从按钮转换来的，提供了平滑的过渡状态。用户看到一个明亮的绿色确认屏幕，但不是在加载动画之前：用户在最终确认之前实际上等待了一两秒钟，但是这种延迟几乎是感觉不到的。

交互

通过实践来学习，这是一句古老的格言并且非常正确。当用户与 UI 进行交互时，他们可通过单独查看每个功效来形成更有意义的结构意识。

不是简单地选择一个项目并使其在观察者面前进行转换，当用户能感受到状态的变换时，可以强化 UI 状态之间的互联。推荐观看一个由 Mary Lou（Manoela Ilic）在 Codrops 上实现的非常好的拖放交互（*http://tympanus.net/Development/DragDropInteractions/*），如图 5-6 所示。

54 作为一个用户，你知道物件被放在哪里，你知道在哪里可以检索。在底部，对吧？事实上没有底部，也没有抽屉，这只是一个 div。但是，因为我们以一种类似占有空间的方式构建动画和交互，并且通过模仿用户熟悉的现实世界进行交互（一个橱柜抽屉），就可以创建一个用户觉得很容易掌控的空间。

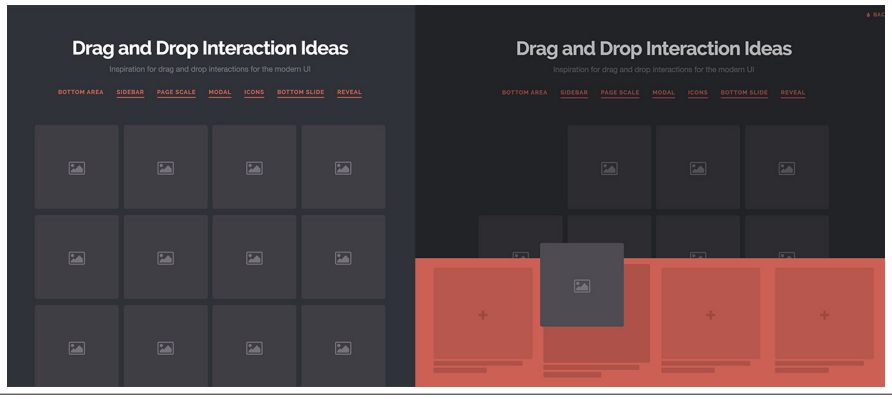


图5-6：我们将元素一致地移动到同一个抽屉里，这个抽屉可以从下面取出

节约空间

当我们使用动画来隐藏和显示页面上非持久化的信息时，能够在有限的空间中，为用户提供更多的信息，包括更多的访问、更多的工具和更多的控件。这变得越来越重要，因为我们建立了一个响应式环境，需要处理大量资料的涌出，而不能让用户感到混乱。

参考图 5-7，这张屏幕截图是一个在页面上节省空间的例子（*http://codepen.io/sdras/full/Kwjyzo/*）。我们可以满足移动端所需的较大触摸点，同时在不需要时将导航折叠在较小的空间中。这个导航是用 Sara Soueidan 的 Circulus 工具（*https://sarasoueidan.com/tools/circulus/*）构建的，它创建了一个 SVG 圆形动画导航。

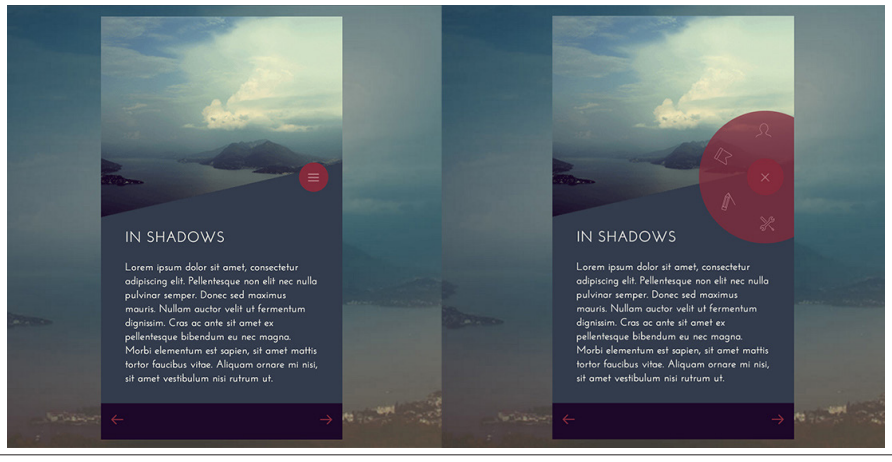


图5-7：使用Sara Soueidan的工具，可以通过隐藏带有动画的部分来节省空间，直到它们被唤起

55

总结

这些动画理论和概念在组合使用时效果最佳。而对于 SVG 动画场景永远都没有最佳的解决方案，使用 SVG 进行动画创作是一项非常重要的能力。当理解了上述核心概念后，我们就获得了所有制作动画的基础认识，自然而然就可以写出代码了。

变换的图标

我们已经讨论了在 UI/UX 模式中有关动画的各种“为什么”，下面让我们来看看“怎么做”。接下来的这个示例基于一个常见的场景来构建，你可以了解到它是如何一步步创建交互效果的。按照同样的方式，我们将学会如何解构一个类似的简单交互场景，并且使用 SVG 将其构建到我们的站点中，不过这并不意味着我们会始终使用这一种方法来完成整个过程。

图标是一种很好的向站点添加简单、有用和信息性的动画的方式，这类动画“小而美”。如果它太冗长或太花哨，不会为用户带来好处，反而会分散用户的注意力。

这种类型的交互不应该让人觉得太冗长。一种常见的做法是将过渡时间保持在 0 到 300 毫秒之间，只要让用户感觉这种过渡不像瞬间发生的即可。

要牢记的另一点是，用户反复看到的 UI 或 UX 模式都应该是小而美的，这样用户就不会觉得一遍遍看起来很累。

56 在我们的例子中（*http://codepen.io/sdras/pen/BKaYyG*），将做一个针对搜索框的放大功能。图 5-8 和图 5-9 显示了其开始和结束的状态。

图5-8：放大镜起始状态

图5-9：结束状态，放大镜图标被单击后杆成为输入区域，放大镜的圆圈变为关闭形状

我们要把放大镜把手变成线条，并将圆圈变成 x 的形状。

在这个例子中，我们会在事件触发时展示输入框。这个简单的 UI 动画中有几个元素进行了移动，所以动效规划对于效果来说非常有帮助。

首先要把焦点放在变长的放大镜把手上，让我们考虑各个状态之间发生的情况。杆本身 57
必须变长，要稍微旋转一些角度，并且必须变换到适当的位置。

让我们通过延长 viewBox 来适应杆大小的变化。从初始化 SVG 开始：

```
<svg class="magnifier" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 32 34">
    <circle class="cls-1" cx="12.1" cy="12.1" r="11.6"/>
    <line class="cls-1" x1="20.5" y1="20" x2="33.1" y2="32.6"/>
</svg>
```

调整 viewBox：

```
<svg class="magnifier" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 300 34">
```

我们还要确保 SVG 在 CSS（SCSS）中初始化好位置，为将来的转换和返回做好准备：

```
.magnifier {
    line {
        transform: rotate(0deg) translateY(0px);
    }
    circle {
        transform: scale(1);
    }
}
```

可以通过几种方式来改变状态：在jQuery中，可以使用一个简单的类的操作；而在React中可直接通过调用`getInitialState()`来获取状态，并通过事件处理器来设置状态。在本书成书时，大部分读者都对jQuery比较熟悉，因此我用它来做示例，不过后面的章节中也会接触到React。我们将使用jQuery 3（使用1.x或2.x也可以），因为它支持SVG上的类操作。

我们需要做的是通过改变`x2`属性的值来更新线的长度。在这个例子中，我们把它从`33.1`延长到`300`：

```
$( document ).ready(function() {
    $(".main").on("click", function() {
        var magLine = $(this).find(".magnifier line"),
            mainInput = $(this).find("input");

        if ($(this).hasClass("open")) {
            $(this).removeClass("open");
            magLine.attr("x2", 33.1);
            mainInput.blur();
            mainInput.val("");
        } else {
            $(this).addClass("open");
            magLine.attr("x2", 300);
            mainInput.focus();
        }

    });
});
```

58

还要在单击按钮时让输入框得到焦点，以便屏幕阅读器被引导到搜索功能的输入端，并将焦点从退出位置移除。如果用户关闭了搜索框，我们希望选择也被清除。这时候，之前的代码已经把线加长，但由于没有进行旋转和变换，线超出了可视区域。让我们用CSS来解决这个问题：

```
.open .magnifier {
    line {
        transform: rotate(-2.5deg) translateY(14px);
    }
    circle {
        transform: scale(0.5);
    }
}
```

SVG DOM 元素的 CSS 变换

当对 CSS 和 SVG 进行变换的测试时，你可能注意到跨浏览器的稳定性会随着运动复杂性的增加而变得难以处理，特别是使用 `transform-origin` 调整变换原点时。这是我使用 GreenSock 开发的一个主要原因。GreenSock 不仅能使你的 SVG 动画稳定可控，还修复了一些跨浏览器时与规范中相悖的变换叠加问题。

在这里不需要用一个完整的 CSS 动画关键帧插值来实现，因为它只是从 A 点到 B 点，所以只需要使用一个 `transition`。我们也会在 SCSS 中使用几个自定义缓动，将其作为变量来重用。

一个很有趣的技巧是，使用 ease-out 缓动函数对于进入场景效果比较好，而使用 ease-in 类缓动函数对于退出场景比较好。考虑到这一点，我们将使用 quad 缓动函数来实现：

```
$quad: cubic-bezier(0.25, 0.46, 0.45, 0.94);
$quad-out: cubic-bezier(0.55, 0.085, 0.68, 0.53);

.open .magnifier {
    line {
        transition: 0.65s all $quad;
        transform: rotate(-2.5deg) translateY(14px);
    }
    circle {
        transition: 0.35s all $quad;
        transform: scale(0.5);
    }
}
```

你会注意到我们在进入状态下使用了入口动画。这部分看起来有点滞后：`.open` 动画将被视为入口动画状态，而退出动画则应该被添加到初始属性。这可能有点违反直觉，你多实现几次就会习惯。折叠的退出动画会有更多的视觉感，并且我们会让动画速度快一些，因为在这个场景中，动画消失得越快感觉越好。

59

```
.magnifier {
    line {
        transition: 0.25s all $quad-out;
        transform: rotate(0deg) translateY(0px);
    }
    circle {
        transition: 0.25s all $quad-out;
        transform: scale(1);
    }
}
```

接下来让我们处理 circle 和 x-out 类。在这个例子中，我们已经在 SVG 上添加了 x-out 类，并对其设置了正确的定位，其实也可以把定位设置在初始的 SVG 上。我没有这样做，因为当我最初创建动画时并不知道动画应用的元素在哪里。在创建功能时保持独立，让我在开发迭代中保留了更多的灵活性。如果你已经有了一份规范的设计语言，它会对 SVG DOM 中的元素提供更好的跨浏览器兼容性。

分离这个元素的另一个原因是 `transition-origin` 值。如果使用的 SVG 结构比较复杂，那么这个值会非常难以定义，但当把一条线封装到一个独立的 SVG 中时，我可以很容易地将其设置为 `50% 50%`，这指的是 x 的中点：

```
.x-out {
    width: 6px;
    padding: 5px 6px;
    transition: 0.5s all $quad;
    cursor: pointer;
    line {
        stroke-width: 2px;
        opacity: 0;
        transform: scale(0);
        transform-origin: 50% 50%;
    }
}

@-moz-document url-prefix() {
    .x-out {
        padding: 5px !important;
    }
}

.open .x-out line {
    opacity: 1;
    transform: scale(1);
    transition: 0.75s all $quad;
}
```

在后面的章节中，我会介绍 GreenSock 的一些强大功能，它可以很方便地处理和定义 `transform-origin` 的值，但 CSS 由于跨浏览器的问题，目前只能轻量处理。

最后可以看到，我们需要添加一个输入框来让例子真正动起来。通过使用一些绝对定位，

可确保 SVG 和输入框处于相同的高度：

```
.magnifier, input, .x-out {
    margin-left: 30vw;
    margin-top: 40vh;
    pointer: cursor;
    position: absolute;
}

.magnifier, input {
    width: 400px;
}
```

然后，我们将确保输入框不包含默认的浏览器样式，但字体大小与 SVG 的大小相匹配：

```
input {
    font-size: 35px;
    padding-left: 30px;
    font-family: inherit;
    color: inherit;
    background: none;
    cursor: pointer;
    box-shadow: none;
    border: none;
    outline: none;
}
```

经过一系列设计，图 5-10 所示的是最终结果。注意在操作时检查这个特定的动画是否正常触发（*http://codepen.io/sdras/full/BKaYyG*）。

图5-10：最终效果

如果你在 SVG 中对整个路径进行变换，请查看第 10 章，因为 JavaScript（特别是 GreenSock 的 SVG）是实现这类效果的最佳选择。不过不需要任何额外的库也可以实现一些简单的

61

动效。

当然，这只是处理一个 UX 模式的一种方法。你会发现大多数用户体验模式，通常都会采用这种解决问题的方法与步骤，从而获得一些不错的效果。

有一些开源库已经完成了这种类型的交互，比如 Sara Soueidan 的 Navicon Transformicons（*http://bit.ly/2mAJDdL*）或 Dennis Gaebel 的 fork（*http://www.transformicons.com/*）。如果不要求进行定制，可以参考一下这些实现。

第 6 章

动态数据可视化

数据可视化是用来呈现不同信息的一个非常有用的方法。幸运的是，由于一些相关的库的流行，比如 D3（*https://d3js.org/*）、Chartist（*https://gionkunz.github.io/chartist-js/*），让我们能够很容易地创建一些动画。并不是说有这些库就可以实现数据可视化，我只是从我工作中接触到的库中选出了我最喜爱的。

在本章中，我们将利用 D3 和 Chartist 来实现数据可视化。Chartist 在刚发布的时候，利用的是已经废弃的 SMIL 来做动画，所以，我不建议大家使用它原生的动画函数。D3 也提供了一些原生动画函数，但是你可能发现，根据 CSS 的一些实现，在屏幕上实现数据可视化并且动态化，应该更容易。

Chartist 与 D3 的优缺点

在 Chartist 中，可以很容易地创建响应式的图表和图像，它非常容易上手。Chartist 封装了 SVG，所以一些 JavaScript 的功能实现起来有点困惑和不直接。考虑到这一点，我强烈建议使用 Chartist 去画简单的图像，并用 CSS 做简单的动画。

而 D3 并不是特别容易上手，但是，它非常易于扩展。你可以利用 D3 创建你想创建的，并且实现很多漂亮的 Web 可视化数据。

简而言之，这并不是固定的答案。你应该选择适合你工作流和网站实现的库，或者你喜欢的库。

教授怎样具体使用 Chartist 或者 D3 去创建图表和图像，已经超出了本书的范围，但是 Chartist 有很棒的实例、文档，并且 O'Reilly 也有很棒的图书对如何学习 D3 做了介绍，如由 Scott Murray 编著的 *Interactive Data Visualization for the Web*，我使用这本书学习了 D3 的技术并且强烈推荐给你。

为什么要在数据可视化中使用动画

动画在数据可视化中非常有帮助，因为其常常作为数据结果的一种表现形式。动画有很多方式来帮助数据可视化：

- 过滤
- 排序
- 演示
- 随时间变化的演示
- 透明度的变化

在第 5 章中，我们讨论了对用户而言保持上下文的重要性。通过重组数据来改变其内容，其中数据过滤可以保持元素的一致性。

参考图 6-1 所示的数据可视化。纽约时报在不同的上下文中呈现了一致的数据，这让读者能够通过有效和多维的方式来处理信息。当呈现的区域保持不变时，用户能够通过在不同上下文中获得更大的信息量。

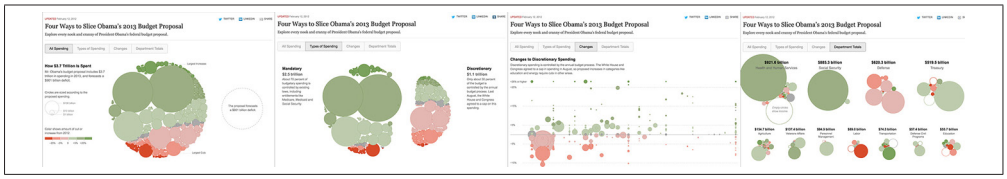


图 6-1：纽约时报通过 4 种不同的方式，重组一样的信息，给予了这些信息新的含义，并且通过动画连接不同的视图来保持不同状态上下文的一致

65 即使是最有效的信息，如果没有展示出来也是没有意义的。[注 1] 这也是为什么动态展示在数据可视化中是非常重要的原因。

66 # 使用 CSS 动画的 D3 示例

D3 中有许多漂亮的块，在开始时你可以直接使用它们或做相关修改（参见图 6-2）。这些块是一些展示例子，里面包含代码和 D3 的实现细节。注意，这些块并不是库的一部分，它们只是贡献者提供的独立的例子，并且版权和版本可能会不同。

要想像已经做好的示例一样漂亮，你可能需要给你的网页添加一些样式使它们更生动。

注 1　Emma Whitehead 和 Tobias Sturt 在 2014 年的 Graphical Web 上提供了“用数据可视化来讲故事”（*http://bit.ly/2lSLKtl*）的文章。

对于这个示例，我选了美国的 3000 家沃尔玛位置的地图。

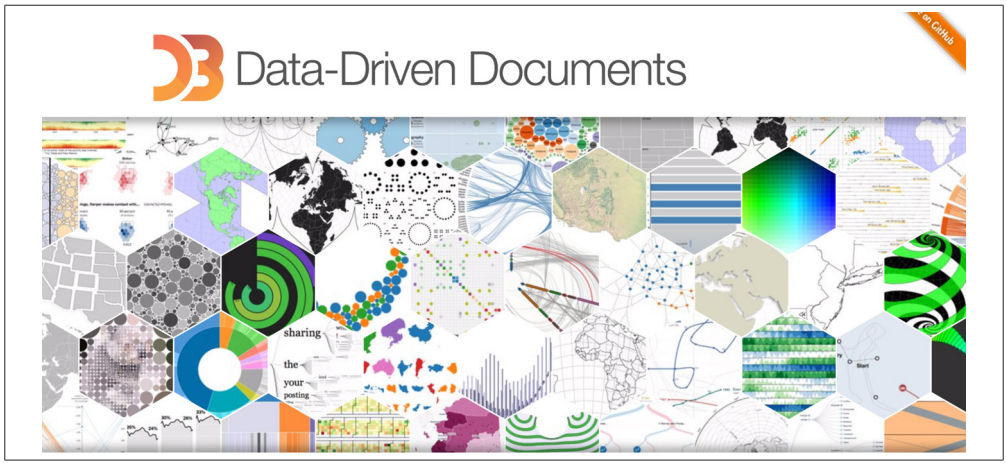


图 6-2：D3首页

用一些样式和简单的 SCSS 函数，我们可以将静态的文档转换为动画效果，完成的动画详见 *http://codepen.io/sdras/full/qZBgaj/*。 67, 68

为了改变基本样式，我们需要不同的类去区分不同的 SVG 的路径类型。在本例中，D3 通过 `.attr()` 函数指定了必要的类名。 69

下面是 JavaScript 代码：

```
svg.append("path")
      .datum(topojson.feature(us, us.objects.land))
      .attr("class", "land")
      .attr("d", path);

svg.append("path")
      .datum(topojson.mesh(us, us.objects.states, function(a, b) {
          return a !== b;
      }))
      .attr("class", "states")
      .attr("d", path);

svg.append("g")
      .attr("class", "hexagons")
      .selectAll("path")
```

下面是 SCSS 代码：

```
svg {
```

```
    position: absolute;
    left: 50%;
    margin-left: -500px;
}

70  path {
    fill: none;
    stroke-linejoin: round;
}

.land {
    fill: #444;
}

.states {
    stroke: #555;
}
```

在每个六边形路径上添加一个额外的类名，让其有动画效果，其实是没有意义的，因为我们可以使用 `nth-child` 选择器来完成。Sass 也可以通过创建一个函数，来构造一个栅格动画。初始的时候，我们将六边形的 `opacity` 设置为 `0`，目的是能够渐变地展现它们：

```
.hexagons path {
    opacity: 0;
}

$elements: 2000;
@for $i from 0 to $elements {
    .hexagons path:nth-child(#{$i}) {
        $per: $i/50;
        animation: 2s #{$per}s ease hexagons both;
    }
}

@keyframes hexagons {
    100% {
        opacity: 1;
    }
}
```

这里只需要不多的代码就可以实现非常漂亮和令人激动的渐变动画。如果想了解更多时间轴的动画效果，可以参考第 12 章。在第 12 章中，我们将会结合拖动实例，通过创建具有交互性和渐变性的动画来讲解 GSAP 时间轴动画。

使用 CSS 动画的 Chartist 示例

让我们也来做一个简单的 Chartist 例子。首先，我们基于一个完整的折线图表来制作，其样式已经满足我们的需要，最有趣的是其中的线条动画。这能够清晰地显示数据本身，并且能更简单地处理数据。

通过 Illusion 来绘制 SVG 的图形，借助 `.getTotalLength()` API 来获取需要知道的 SVG 路径的长度。[注 1] 71

```
setTimeout (
    function() {
        var path = document.querySelector('.ct-series-d path');
        var length = path.getTotalLength();
        console.log(length);
    },
3000);

// 输出
a: 1060.49267578125
b: 1665.3359375
c: 1644.7210693359375
d: 1540.881103515625
```

我们将使用这些数据让路径有一定的动效。比如，通过 CSS 让这些路径一点一点地绘制。

首先，给其中的一条路径设置 `stroke-dasharray` 属性：

```
.ct-series-a {
    fill: none;
    stroke-dasharray: 20;
    stroke: $color1;
}
```

效果如图 6-3 所示。

我们可以给 `stroke-dasharray` 属性设置无限制的长度，同时也可以设置一个 `stroke-dashoffset` 属性，它的值也可以随意设置。`stroke-dashoffset` 属性的含义和它的名字一样，就是设定虚线段的间隔值。这个属性其实就是动画属性之一。 72

注 1　对于 SVG 路径完整的操作可以查阅 MDN（*https://mzl.la/2lkvTlG*）上的资源。

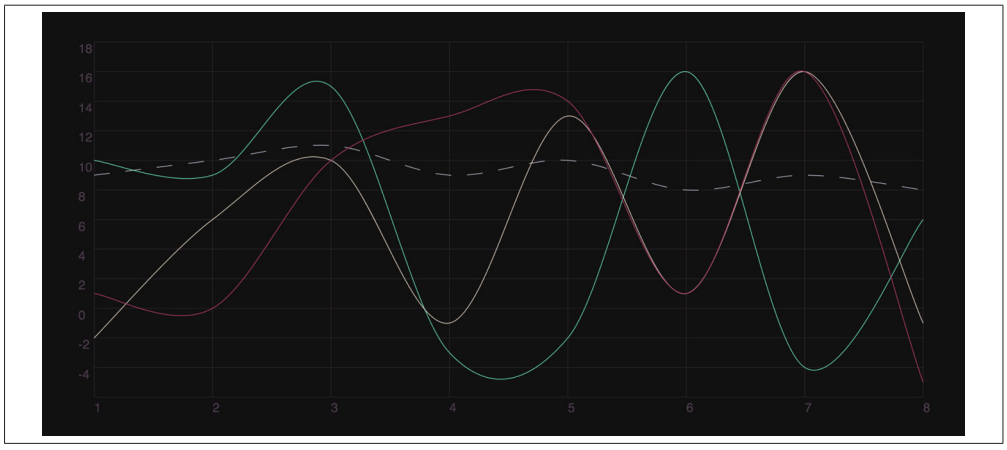


图6-3：带有 stroke-dasharray 属性的路径

现在可以使用控制台输出和我们的表格来创建一个动画，该动画会使用整个路径长度并使其发生偏移。这可以让数据（*http://codepen.io/sdras/full/oxNmRM*）看起来是在 viewBox 中绘制的（参见图 6-4 和图 6-5）。我们打算多次使用相同的信息，所以可以使用 mixin 来进行复用。同时也可以给 dashoffset 和 dasharray 设置不同的值。为了统一，将其最终值设为 0：

```
@mixin pathseries($length, $delay, $strokecolor) {
    stroke-dasharray: $length;
    stroke-dashoffset: $length;
    animation: draw 1s $delay ease both;
    fill: none;
    stroke: $strokecolor;
    opacity: 0.8;
}

.ct-series-a {
    @include pathseries(1093, 0s, $color1);
}

@keyframes draw {
    to {
        stroke-dashoffset: 0;
    }
}
```

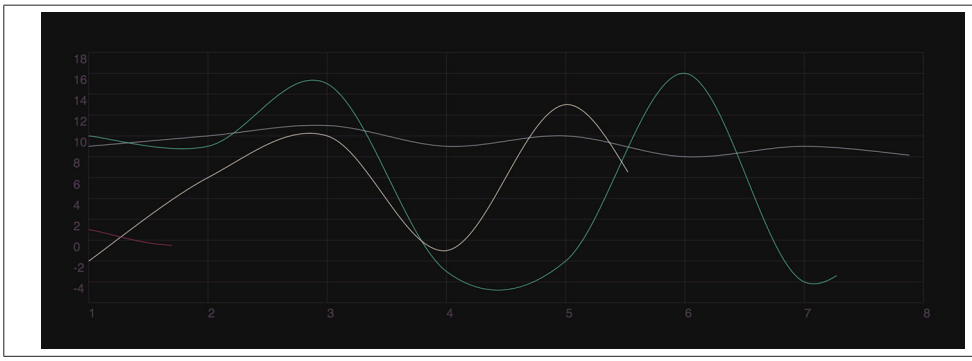


图6-4：可以看到渐变式线条的绘制

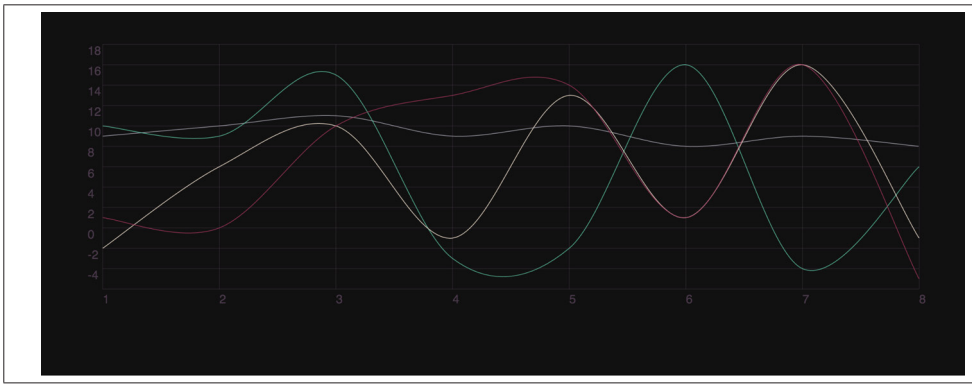


图6-5：这是它的结果

上面仅仅是创建 Chartist 可视化数据动画的一种方式。你可以在其官网（*https://gionkunz.github.io/chartist-js/examples.html*）找到很多其他的例子。在使用 Chartist 和 CSS 时，编写初始、结束和持续状态的方式，完全由你自己决定。 73

接下来，我们将在后面几章中更深入地学习数据可视化，但是在学习之前还要了解一下 JavaScript 是如何工作的。接下来是一个动画技术的简要对比，后面我们将换一种语言。

用 D3 来做动画

在本节中，在不使用 CSS 的前提下，我们将通过 D3 做一个简单的动画例子。在后面的例子中，我们将使用版本 4（需要注意，版本 3 和版本 4 是不兼容的，如果你使用版本 3，那么下面的例子可能无法使用）。

如果你还记得第 1 章中介绍的内容，那么会发现，一条线其实就是由 x 和 y 坐标平面上

的点绘制而成的，也可以发现这个方法对于一些简单的数据可视化来说非常有用。观察下面的代码，可以用之前 SVG 的知识来理解它：

```
var line = d3.line();
var data1 = [[0, 0], [200, 300], [400, 50], [500, 300],
    [550, 300], [600, 50], [700, 120], [775, 250]];
var data2 = [[0, 100], [220, 120], [300, 250], [500, 10],
    [520, 120], [575, 250], [600, 300], [775, 50]];

d3.select('#path1')
    .attr('d', line(data1))
    .transition()
    .attr('d', line(data2))
    .delay(1000)
    .duration(3000)
    .ease(d3.easeBounce);
```


我们通过 `d3.line()` 方法来设置相关的线条。该方法通过数据中的 x 和 y 两个平面坐标字段来完成绘制。接着，我们可以给线条的两个状态变化添加一个渐变效果。另外，还可以选择是否添加延时、持续时间和变换特效。

之前的代码能够添加线条两个状态的渐变效果，如图 6.6 所示。

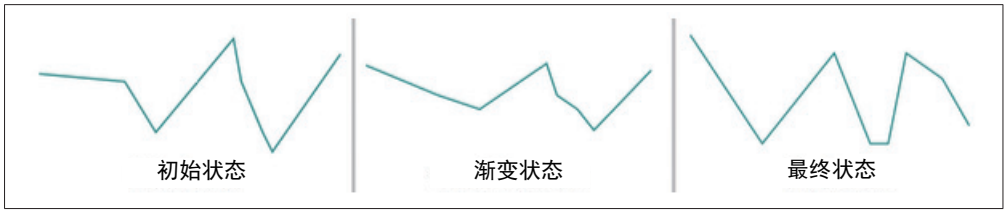


图6-6：线条的三个状态

你可以对其他属性复用该方法，比如 `colors`、`coordinates`。D3 对于该方式是非常灵活的。

不同数量路径点的动画

虽然 D3 很灵活（它可以支持 SVG 的大部分内容），但是 SVG 对于不同动画的路径值有非常严格的要求。如果第二个数据集相比于第一个有不同的长度，渐变的效果看起来会非常奇怪和丑陋，或者会直接失败。这也是为什么 GreenSock 的 MorphSVG 非常好用的原因，因为它对于这种情况能处理得很好。d3-interpolate-path（*https://github.com/pbeshai/d3-interpolate-path*）是一个独立于 D3 开发出来的库，用来创建一些漂亮的路径动画。这里有一篇很好的博客介绍它（*http://bit.ly/2lSOMOz*）。

使用 D3 创建摇摆动画非常容易，而且可以与 CSS 一起使用你给每个元素计算出的各自的延时。如果你对 JavaScript 的 `for` 循环很熟悉，那么该方法的应用对你来说应该不是太大的问题：

```
transition.delay(function(d, i) { return i * 10; });
```

如果我们在颜色中用的也是这种插值方式，那么代码看起来如下所示（我们将上面的代码示例更新为一个散点图，让颜色更容易被看到）：

```
var dataset = [ 5, 17, 15, 13, 25, 30, 15, 17, 35, 10, 25, 15],
    w = 300,
    h = 300;

// 创建 SVG
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)
            .attr("height", h);

// 通过数据在 SVG 中创建相应的形状
svg.selectAll("circle")
    .data(dataset)
    .enter()
    .append("circle")
    .attr("class", "circles")
    .attr("cx", function(d, i) {
     return 10 + (i * 22)
     })
    .attr("cy", function(d) {
        return 200 - (d * 5)
    })
    .attr("r", function(d) {
        return (d / 2)
    })
    .transition()
        .style("fill", "teal")
        .duration(1500)
        .delay(function(d, i) { return i * 100; });
```

75

完整示例可以参考 CodePen 例子（*http://bit.ly/2fpuPe3*）。

链式和重复

对于更复杂的效果，我们可以添加多个渐变效果，甚至创建循环。为了使用链式动画，我们像之前一样在两个状态之间添加 `.transition()` 方法，不过，为了让整个动画循环，还需要更改一点语法，使用递归的方法。下面是具体代码：

```
.transition()
    .on("start", function repeat() {
        d3.active(this)
            .style("fill", "purple")
          .transition()
            .style("fill", "blue")
          .duration(2000)
          .transition()
          .duration(2000)
            .on("start", repeat);
      });
```

详细例子可以参考 CodePen（*http://bit.ly/2goB8mh*）。

76 不过，如果你打算写一个非常复杂的链式或者重复动画，推荐使用 GreenSock 来做。我们在后面几章中会介绍 GreenSock。你会发现，上面的代码能够与 D3 的输出完美结合，并且提供精确的时间轴和顺序控制。

第 7 章 77

Web动画技术大比拼

之前的内容仅涵盖了使用 CSS 来实现各种动画。从这一章开始我们将转向 JavaScript，不过开始之前要做一件很重要的事情：对比各种网页端实现动画的技术并找出它们各自的优缺点，以便大家更了解正在使用的技术，并选择其中最适合当前工作的工具。

在本章的末尾，我们也将讨论与 React 配合使用的工具，由于虚拟 DOM 的原因，在 React 环境中运行动画与通常方式有所不同。

没有办法覆盖所有的动画库，所以通常我会推荐那些更令我感兴趣的。请记住，这些建议主要基于我自己的经验，你可能对这些选择抱有不同看法，这是没问题的。

TL;DR

可以在下面深入阅读各种实现方式的利弊及对比，不过基于我长期使用的经验，这里给出一个简短的建议：GreenSock，它处理了 SVG 的一些跨浏览器问题，并且其实现考虑了各种动画场景，因此 GreenSock 会成为我最常用于生产环境的动画技术。

原生动画 78

在谈论动画框架（库）之前先来看一些原生实现方式。大多数动画框架都会使用原生动画技术，所以对其了解越多，越能理解动画的抽象是如何实现的。

CSS/Sass/SCSS

把 CSS 提到这么靠前的原因是，CSS 体现了 Web 动画实现方案的奥卡姆剃刀原则 —— 当所有技术都可以实现时，最简单的解决方案是最好的，特别是当你需要快速搭建一个

动画效果时，可以使用 CSS 动画——通过一组关键帧在不同状态之间进行转换来实现。

优点：

- 不需要依赖外部框架。
- 性能很好。预处理器（如 Sass 和 LESS）允许利用函数生成 `nth:child` 伪类来产生令人吃惊的效果。对于缓动（easing）函数等也可以使用变量，从而让动画保持一致性。
- 可以使用原生的 JavaScript 监听 `onAnimationEnd` 以及其他与动画相关的事件（*http://codepen.io/sdras/full/PqXeMX/*）。
- 沿路径运动这样的效果将会在 CSS 中被支持，这样的能力对于实现真实的运动动画是很有必要的，而由于 SMIL 渐渐被抛弃，这一点已经变得越发重要了。

缺点：

- Bézier 缓动效果（*http://bit.ly/2lPNF3G*）在某些情况下会受到限制。由于 Bézier 缓动只能通过两个控制点来控制曲线，无法产生复杂的物理效果，如反弹或弹性振动，而这些对于实现逼真的动画效果（但非必需的）是非常重要的。
- 如果需要依次运行三组以上的动画，我建议使用 JavaScript。CSS 中对动画的运行顺序是通过延迟来控制的，这是很复杂的。当你调整时间的时候，不得不重新计算所有的延迟。可以通过监听原生 JavaScript 事件来解决这个问题，但这样在编码时会切换不同语言，也不是很好的方案。持续时间长、复杂、有运行顺序的动画更容易在 JavaScript 中编写和阅读。
- 对沿着路径运动效果的支持还不是很好。你可以投票支持 Firefox（*https://mzl.la/2lSTls5*）。我注册了一个 bug 报告（*http://apple.co/2kWpOQN*），并要求 CSS 中的运动路径模块要作为一个功能。
- CSS + SVG 动画会有些奇怪的问题。一个例子是在 Safari 浏览器中，动画中同时使用不透明度和变换可能会失效或产生奇怪的影响。另一点是在 Web 规范中，对于 CSS 变换原点的定义并不是人们通常理解的那样，从而当变换应用到元素上时会产生预期之外的效果。希望 SVG2 能解决这个问题，而现在对于移动端的 CSS 和 SVG 动画，有时需要一些特殊技巧才能保证动画符合预期。那些使用 CSS 的动画框架，在处理大量效果的同时（例如 GSAP），可以对其进行修正。
- 编写 CSS 动画时需要声明关键帧，然后编写元素本身的 CSS 来使用动画。这意味着运行动画所需的代码需要维护两个地方。有时这样很好，方便对动画的复用。但大多数情况下，这意味着可读性被破坏，因为你必须在两个地方更新代码。

requestAnimationFrame()

`requestAnimationFrame()`（简称 rAF）是 JavaScript 中 `window` 对象上的原生方法。这个方法非常赞，在 JavaScript 引擎的控制下，它确定了最适合当前环境的动画帧速率，并且只会以这个速率运行。例如，在移动设备上不会像在桌面设备上那样使用高帧率。当浏览器未获得焦点时，动画也会停止运行，以免不必要地占用系统资源。因此，`requestAnimationFrame()` 是一个性能非常好的动画实现方式，后面我们讨论的大部分框架都在内部使用了它。

`requestAnimationFrame()` 以类似递归的方式进行工作，它在绘制动画队列的下一帧之前会执行逻辑，然后再次调用自己，继续执行。这可能听起来有点复杂，但这样的实现方式意味着你有一系列不断运行的命令，因此它的插入可更好地呈现中间步骤。

在第 15 章我们会讲解更多有关 `requestAnimationFrame()` 的知识。

canvas

尽管 canvas 基于栅格图形（*http://bit.ly/2lQqDKt*），而 SVG 基于矢量图形，你仍然可以在 canvas 中使用 SVG。由于 canvas 是基于栅格的，SVG 可能不能像以前一样什么都不用管，而是需要做一些处理。

```
var canvas = document.querySelector('canvas'),
    ctx = canvas.getContext('2d')
    pixelRatio = window.devicePixelRatio,
    w = canvas.width,
    h = canvas.height;

canvas.width = w * pixelRatio
canvas.height = h * pixelRatio

ctx.arc (
    canvas.width / 2,
    canvas.height / 2,
    canvas.width / 2,
    0,
    Math.PI * 2
)
ctx.fill();
canvas.style.width = w + 'px';
```

80

以上效果不需要太多的代码，但是当你习惯 SVG 是跟分辨率无关后，这里会造成一些困扰。Egghead（*http://bit.ly/2moovaE*）上有一个很棒的视频讲解了如何处理这个问题。

我在这里没有使用 SVG，但我了解到 Tiffany Rayside（*http://bit.ly/2laHqnp*）和 Ana Tudor（*http://bit.ly/2lkYFDy*）在 CodePen 上放了一些很赞的作品，推荐你去他们的个人主页探索一下。

Web 动画 API

Web 动画 API，用于浏览器和开发者使用原生 JavaScript 来描述 DOM 元素上的动画效果。通过这个 API 可以创建更复杂的序列动画，而无须加载任何外部脚本（注意，现在可能需要使用 polyfill）。这个 API 是通过提炼各种各样的动画框架，以及开发者使用 JavaScript 实现动画的最佳实践而创建出来的。Web 动画 API 将 CSS 动画的高性能，以及 JavaScript 控制动画运行的灵活性整合在了一起。

优点：

- 实现序列动画很简单，代码清晰易读。Dan Wilson 通过一个很好的例子比较了 CSS 关键帧和 Web 动画 API（*http://codepen.io/danwilson/pen/QwrZwd*）。
- 性能表现似乎很好，不过我还是建议你使用自己的性能测试来看一下。

缺点：

- 在本书出版时，浏览器对这个 API 的支持并不是很好。目前已经有比较好的 polyfill 支持它，但它仍在变化中，所以直到规范达到最终版后，我才会谨慎地在生产环境中运行它。尽管如此，它仍然是 Web 动画的未来，因此在这段时间可以尝试。
- GSAP 中的时间轴动画拥有更多强大的功能。而对我来说重要的是 SVG 跨浏览器的稳定性，以及通过一行代码即可运行一系列动画的能力。

第三方框架

GreenSock（GSAP）

GreenSock 目前是网上最复杂的动画库，我非常喜欢使用它。对这个框架的偏爱来自于日常工作和实践，以及在浩如烟海的动画库中屡屡碰壁的血泪史。我尤其喜欢使用这个框架来操作 SVG（*http://bit.ly/2kWAyPa*）。GreenSock 的动画 API（*http://greensock.com/*）拥有很多非常优秀的特性，同时 GreenSock 也有完善的文档（*http://greensock.com/docs/#/HTML5/*）和论坛（*http://greensock.com/forums/*）用于查阅。

优点：

- 对于非原生动画支持（*http://bit.ly/2lGE4fq*）的东西来说，它的表现非常出色，这是非常值得一提的。
- GSAP 时间轴的很多功能比当前 Web 动画 API 的实现更为强大：对于我来说，更关注的是 SVG 的跨浏览器稳定性，以及在 SVG 中通过一行代码管理大量动画执行的能力。
- 如果想实现一些花哨的效果，比如文字动画、SVG 变形，那么 GreenSock 也有大量其他插件来支持。
- GreenSock 的 BezierPlugin（*https://greensock.com/BezierPlugin-JS*）提供的路径运动能力，是目前对路径运动支持最好的。
- 如前所述，它解决了 SVG 跨浏览器的困扰，因此特别适用于移动端。
- GreenSock 的可视化缓动编辑器（*http://greensock.com/ease-visualizer*）提供了非常逼真的方式来创建缓动函数。它甚至允许通过一个 SVG 路径来创建自定义缓动。
- 由于 GreenSock 可以对任意两个整数区间做补间动画，因此你可以实现一些很酷的效果，如对 SVG 滤镜实现动画，可获得一些令人惊叹的效果（*http://codepen.io/sdras/full/gaxGBB/*）。对于 GreenSock 来讲，天空才是极限。关于这点会在第 15 章进行更多介绍。

82

缺点：

- 使用插件时必须为其许可证付费。不过在购买之前可以尝试 CodePen 上的开放版本。
- 管理第三方依赖时，必须关注目前生产环境中正在使用的版本。因为框架的新版本会定期出现，升级时需要进行测试（这是使用任何框架都会遇到的事情）。

mo.js

mo.js（*http://mojo.io/*）是由 LegoMushroom 组织（*http://codepen.io/sol0mka/*）维护、Oleg Solomka 写的一个动画框架。Oleg Solomka 是一个非常有才华的动画制作师，他已经为这个框架做了很多优秀案例，这让我非常兴奋。这个框架仍处于测试阶段，不过现在已经接近发行阶段了。有关如何使用它的更多细节，请参阅第 13 章。

优点：

- 像形变、爆发和旋涡这样的效果，你可以开箱即用——所以你不需要成为世界上最好的或最具创意的插画家就可以实现这些美妙的效果。
- mo.js 提供了一些很好的工具，包括运行对象、时间轴和自定义路径编辑器。这

本身就是使用它的最强有力的理由之一。

- mo.js 提供了不同的动画实现方式，你可以使用对象，或在过程式代码中绘制一个动画变换——可以自己决定最适合你的方式。

缺点：

- 它还没有提供使用 SVG 作为父对象自定义形状的能力（我相信 LegoMushroom 正在开发此功能），因此使用坐标系统和缩放来进行响应式开发不太直观，也难于在移动设备上工作。这是一项相当先进的技术，因此你目前可能不需要它。
- 它不像 GreenSock 那样做了浏览器兼容，这意味着跟使用 CSS 一样，你需要编写 hacks。LegoMushroom 表示他们已经在努力解决这个问题了。

83 Bodymovin'

Bodymovin'（*https://adobe.ly/2l8hD4i*）用于在 Adobe After Effects 中构建动画，并可以轻松导出到 SVG 和 JavaScript。Bodymovin' 的一些演示令人印象深刻（*http://codepen.io/airnan/*）。不过我不会使用它，因为我喜欢使用代码从头开始构建东西（所以这违背了我的目的），但是如果你的团队中除了开发人员更多的是设计师，这个工具对你们来讲真的很棒。我唯一可以预料的一件事是，如果以后需要改变这个效果，必须重新导出，所以这个工作流程可能是比较奇怪的。此外，输出的代码看起来像是一团乱麻，不过我还没观察到这些会影响性能，所以使用应该是没问题的。

不推荐使用

SMIL

SMIL（Synchronized Multimedia Integration Language）是一个原生的 SVG 动画规范，它允许直接在 SVG DOM 中进行移动、变形，甚至与 SVG 进行交互。使用 SMIL 存在很多优缺点，但最大的问题会让我转向废弃这个方案：SMIL 正在失去支持。我写了一篇如何将 SMIL 实现的动画迁移到最新技术上的帖子（*http://bit.ly/2lUZS8d*）。

Velocity.js

Velocity（*http://velocityjs.org/*）提供了之前 GreenSock 能够提供的动画序列控制，且简单明快。下面提到的这些缺点，使我放弃使用 Velocity.js 了。 Velocity.js 的语法看起来有点像 jQuery，如果你已经在使用 jQuery，对其 API 上手速度快会是一个很大的优势。

优点：

- 比 CSS 更容易控制多段动画的执行。
- 有许多开箱即用的缓动方法，也支持弹性运动（*http://codepen.io/julianshapiro/pen/hyeDg*）。同时还支持将一系列缓动函数传递到 step-easing 数组中（*http://julian.com/research/velocity/#easing*）。
- 文档很全，易于入门学习。

缺点：

- 效果不是很好（*http://bit.ly/2lGE4fq*）。尽管有一些相反的说法，但当我运行自己的测试时发现有些效果并没有真正实现。不过我建议读者自己测试，因为代码还在更新。
- GSAP 的体积比较大，但能提供更好的性能和跨浏览器稳定性。jQuery 在性能测试中落败，但未来可能会因为内部使用 rAF 技术而改变。Velocity.js 的速度不差，但没什么突出表现。

84

Snap.svg

很多人认为 Snap（*http://snapsvg.io/*）是一个动画库，而实际上并不是这样。我之前对 Snap 进行过性能测试，并且 Dmitri Baranovskiy（这个框架以及前身 Rafael 的作者，拥有令人难以置信的聪明才智）在 SVG Immersion Podcast（*http://bit.ly/2laMDf2*）上提到，对于动画来讲，Snap 不是最适合的工具。在我们的聊天中他说：“要注意一点，Snap 不是一个动画库，当需要实现动画时，我建议使用 GSAP。”

其实，jQuery 并不适合 SVG，尽管它现在支持类操作（*http://bit.ly/2lUZLt5*）。如果需要使用 SVG 进行大量 DOM 操作，Snap 是推荐的工具，这也是 Snap 的正确使用场景。

有一个名为 SnapFoo 的库(*http://yuschick.github.io/SnapFoo/*)将 Snap 的领域扩展到动画。我还没研究过，但看起来很酷。

基于 React 的动画工作流

在讨论 React 之前，首先要看一下为什么在 React 中处理动画时需要区别对待。主要区别在于文档对象模型（DOM），DOM 描述了如何使用对象创建一个结构化文档，并且其结构主要为树形结构。

React 有一个虚拟的 DOM 来抽象整个 DOM 结构。React 这样做有很多原因，对我来说最令人信服的是能够弄清楚什么会导致 DOM 发生变化，并且只更新需要更新的部分。

当然，这种抽象是有代价的，之前用的一些旧技巧在 React 环境中会给你带来麻烦。例如，jQuery 跟 React 在一起就不容易和睦相处，因为 jQuery 的主要功能是与浏览器的原生 DOM 进行交互和操作。但我经常会看到一些奇怪的 CSS 竞争操作。以下是我在 React 工作流中实现动画的一些建议。

React-Motion

Cheng Lou 编写的 React-Motion（*http://bit.ly/2lSUy2l*）被认为是在 React 中实现动画的最佳方法，并且 React 社区几乎已经使用其代替了老旧的 `ReactCSSTransitionGroup`（*http://bit.ly/2lkVbRs*）。我非常喜欢 React-Motion，不过当你使用时要注意一些关键点，如果不了解的话会比较头疼。

85 React-Motion 非常类似于基于游戏的动画（*http://bit.ly/2lQv5Jf*），你可以在其中提供元素的质量和初始物理状态，并将其传递给元素，之后元素就动起来了，它不需要像 CSS 或者其他传统的基于坐标的运动方式那样指定一系列的动画时间。由 React-Motion 实现的动画效果看起来很逼真、很酷炫（*http://codepen.io/sdras/full/pyedJE/*），但有一点比较难处理。如果有两个不同的元素必须在同一时间启动，并同时到达终点，那么你会发现它们很难“齐头并进”。

最近，Cheng Lou 在 React-Motion 中加入了 `onRest`（*http://bit.ly/2lQKuJh*）方法来支持传统的回调运行的方式。尽管如此，它并没有太大的进步，因为这种方式与该项目本身的实现原理是背道而驰的，而且实现一个循环动画也不容易（不是无限循环，无限循环会导致很多问题）。你可能永远不会遇到这个问题，但是我曾经遇到过一次。

优点：

- 动画看起来很美观，弹性效果非常好。
- 具有非常独特的弹性效果——在大多数 JS 库（如 GSAP 和 Velocity）中都可以实现弹性效果，React-Motion 中的弹性效果直接基于最后一个元素的运动，不是移除最后一个元素并重新渲染一个新的，所以在动画运动效果上会比较好。
- 这可能是在 React 中实现动画最好的工具了。

缺点：

- 它不像其他一些库或原生方法一样即插即用，这意味着最终你会写出更多的代码。有些人喜欢这个而有些人不是。对初学者来说，这并不是一件好事。
- 由于代码的复杂性，实现顺序执行动画并不像其他方案那样直观和清晰。
- `onRest` 仍然没有提供交错动画的参数。

在 React 中使用 GSAP

GreenSock 提供了这么多功能，因此它值得在 React 环境中使用。这需要 GSAP 比平时更加轻量，有些事情不需要在 React 中处理（通过 DOM）。也就是说，我已经有了几种不同的使用方式：

- 在 React 组件生命周期中使用（*http://bit.ly/2mf3hih*）。
- 当需要交互时触发其运行。当用户交互时创建一个函数，然后将其挂载到类似于 `onClick` 的事件中。
- Chang Wang 有一个很好的帖子（*http://bit.ly/2lQD7Bz*）讲解了如何将 GSAP 挂载到 `ReactTransitionGroup` 中，这是一个非常好的方式。
- 你可以使用 React-Gsap-Enhancer（*http://bit.ly/2l8hTA9*）等库。当遇到一些复杂的控制动画执行顺序的场景时，React-Gsap-Enhancer 可以帮助你处理，而在通常情况下不需要“杀鸡用牛刀”，可以直接在 React 的生命周期内使用 GSAP。

86

在 React 中使用 canvas

canvas 在 React 中可以正常运行。可以选择完全绕过 DOM，创建一个节点，之后在上面创建所有动画。在 React 中使用 canvas 的方式，跟之前讨论的 canvas 的优点和限制相同（见本章前面“canvas”小节）。也可以将一个 canvas 动画实现为 React 组件，但由于虚拟 DOM 的关系，实现细节可能会变得更加复杂。

有几个很好的框架可以帮助你在 React 中使用 canvas。React-Canvas 是由 Flipboard 团队开发的，他们想在 DOM 上实现 60fps 的动画效果。这个报告已经有段时间没更新了，并且这个项目只关注 UI 元素，因此实现其他类型的动画需要一些额外的工作。

React Konva（*https://github.com/lavrton/react-konva*）是一个有趣的、声明式的使用 canvas 和 React 的方法。它很容易就能创造出好看的图形，但动画能力有些缺乏，开发人员也在文档中承认了这一点，因此如果你愿意提交 pull request（PR），那么你可以帮助改进这个项目。

在 React 中使用 CSS

CSS 会重新进入我们的考虑范围，因为它可能是在 React 中创建动画的最简单的方法。我喜欢使用 CSS 动画来处理诸如 UI / UX 交互的小事情，只不过当尝试使用延时来实现连续的动画时，效果会变得有点奇怪。除此之外，CSS 非常好用，特别是对于小的 UI 调整。

总结

不幸的是，本书不可能深入分析以上所有的方式。由于 GreenSock 功能强大，用途广泛，我们将主要关注 GreenSock 的 Animation API。我们还将介绍 mo.js、React-Motion 和 `requestAnimationFrame()`，以便你了解如何通过“裸金属”的方式使用 JavaScript 完成动画。

第 8 章 87

用GreenSock做动画

在前面的章节中，我们介绍了一些你会选择 GreenSock 作为一个动画库的原因。在本章中，我们会介绍一些如何使用 GreenSock 做动画的基本知识。

即使你更喜欢 CSS，你也同样可以再掌握 GreenSock。当使用 JavaScript 来做动画的时候，你没必要了解它所有的知识。当然，这些更喜欢用 JavaScript 的人接触起 GreenSock 来会稍微快一些，而且可以更简单地进行调试。但是，我确实认为 GreenSock 的语法比较简单，完全可以让一个 CSS 开发者有能力掌握它。在某些方面，GreenSock 比 CSS 更简单：CSS 中的帧动画把帧的定义放到一个地方，和应用它们的 CSS 属性区分开，然而 GreenSock 可以让你在一个地方处理所有的东西。

GreenSock 已经开发了 10 年，它之前是一个 Flash 的工具。这成为它在竞争中一个巨大的支柱，因为设计者们更熟悉开发者遇到的问题。他们都是非常亲切的，有一些固定的人在论坛上帮助大家，所以当你遇到问题的时候，有一个非常好的社区可帮助你。

让我们开始吧!

开始使用 GreenSock

要使用 GreenSock，你需要在页面开始部分中引入 TweenMax，具体代码片段如下所示：`<script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/1.19.0/TweenMax.min.js"></script>`。可以将 `/1.19.0/` 替换为任何一个最近的版本，版本号可以在 *https://cdnjs.com/* 网站上看到。

或者你可以在命令行使用 `yarn` 或者 `npm` 来安装，命令如下：`npm install gsap` 或 `yarn install gsap`。 88

GreenSock 的基本语法

我们将开始制作一个真正的简单示例，结果如图 8-1 所示，代码如下：

```
TweenMax.to(".element", 2, { x: 100 });
```

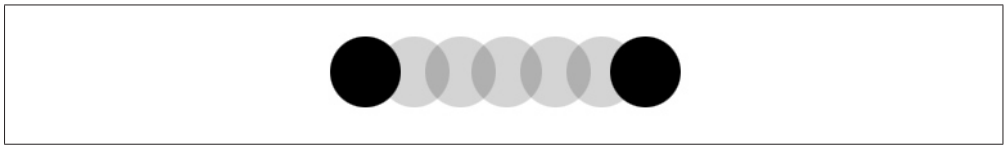


图8-1：我们有一个球，它有一个 .element 类

在这个示例中，球向右移动了 100px。下面我们一点一点分析这里面的语法，然后再考虑一些其他的选项。

TweenMax/TweenLite

```
TweenMax.to(".element", 2, { x: 100 });
```

代码片段一开始的 TweenMax 告诉浏览器：我们将要使用 GreenSock 里的动画 API 代码（它在之前引入的库文件中被加载）。如果选择使用更小版本的库文件，你可以用 TweenLite 来替代。TweenLite 的优势是它非常小，同样的，TweenMax 的优势是它加载了类似循环、CSS 属性（你可能会需要）和 TimelineMax 库，它们扩展了较小的 TimelineLite。这两个是可以互相替换的，且不会改变动画工作的方式。

.to/.from/.fromTo

```
TweenMax.to(".element", 2, { x: 100 });
```

另一个就是 .to 方法，就像你想象的，它告诉元素去改变状态。

你也可以用 .from 方法，它代表元素源自你任意在大括号里指定的动画对象，可以更改它的默认值。或者可使用 .fromTo，它可以让你更细颗粒度地控制从哪里开始、从哪里结束。

89

.fromTo 对动画很有用，它会被重新触发。举例如下，你触发一个动画，它会缩放 50%。

然后你又让它缩放 50%——但是它已经在那了。这个动画在第二次触发之后将会看起来什么都没有做。

当我们使用 .fromTo 时，这个语法看起来有一点不一样：

```
TweenMax.fromTo(".element", 2, {
  x: 0
 }, {
  x: 100
 });
```

你可以看到，我改变了一些代码，这样看起来更清晰，元素会在 *x* 轴从 0 到 100 移动。

Staggering

还可以使用 .staggerTo、.staggerFrom 或者 .staggerFromTo。这会有同样的动画效果，以多种层叠的方式重复应用到一组你设计的对象中。使用 SVG，我发现放置元素到一个组里很方便。为了完成它，我们给这个组加了一个类。举个例子，在下面的代码中，动画效果会应用到含有 .element 类名的组内部的所有 circle 上：

```
TweenMax.staggerTo(".element circle", 2, {
  x: 100
}, 0.1);
```

这个代码片段展示了我们改变了的内容：用 .staggerTo 替代了 .to，在代码块的最后增加了一个额外的参数 0.1。它控制每一个 stagger 之间的时间。我们还指向了所有含有 .element 类名的组里面的 circle。具体效果如图 8-2 所示。

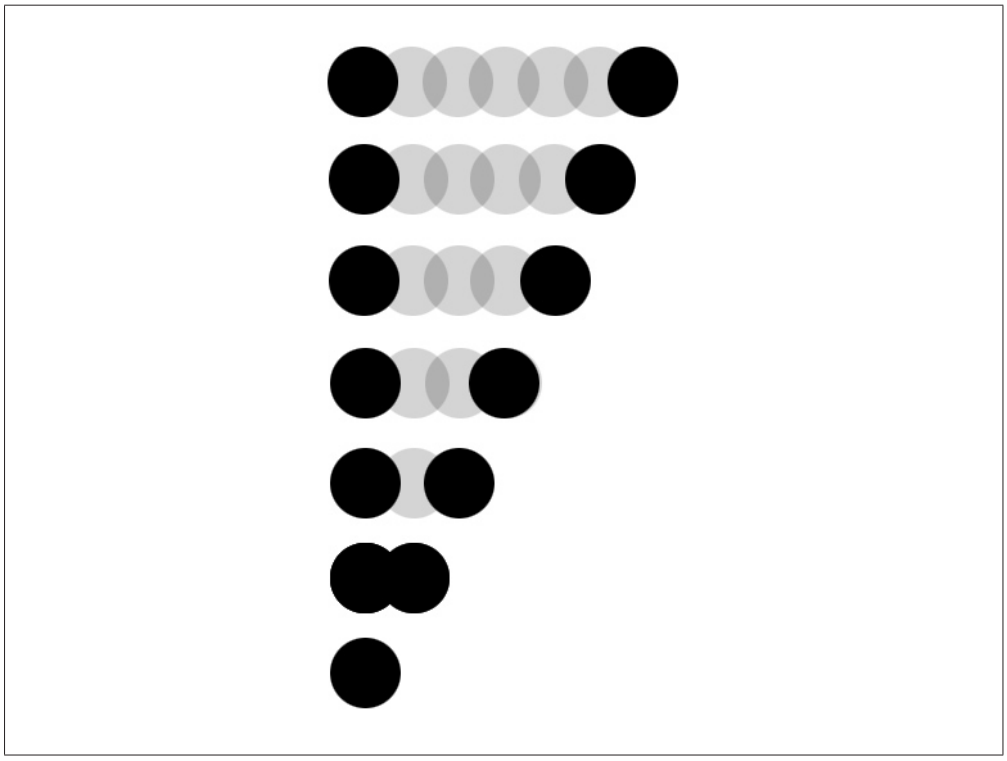


图8-2：所有的球都以相同的值运动，但是一个接着一个，有时间间隔

90

反方向的 stagger

如果你想从最后一个元素开始到第一个元素应用 stagger，非常简单。给间隔设置一个负值（这里设置了值 `-0.1`）即可：

```
TweenMax.staggerTo(".element
    circle", 2, {
  x: 100
}, -0.1);
```

还有更多 `stagger` 高级的类型可用，包含使用 `cycle` 属性、随机的 staggering 值等。你可以在第 11 章获取更多相关的信息。

element

```
TweenMax.to(".element", 2, { x: 100 });
```

GreenSock 关联目标元素的方式和原生 JavaScript 中的 querySelector() 或者 querySelectorAll() 很像，甚至和 jQuery 选择器的行为更接近。在这里你可以传递一个或者多个元素，它们可以是类名、ID 或者是属性（像 path、circle 或者 rect）。无须担心节点列表，它们都会被抽取，这样让 DOM 和跨浏览器支持更简单。

你可以使用带引号的选择器字符串，像上面的示例一样，直接指向元素，但是 GreenSock 也接受变量（比如，var el = document.querySelector(".el")）。当针对一个元素多次使用时，我更倾向使用这样的变量，以避免重复和多次查找。

Duration

```
TweenMax.to(".element", 2, { x: 100, delay:
    2 });
```

这应该是我们见过最简单的值了。我们传递了一个整数，它代表动画运行多久。2 代表 2 秒，0.3 代表 0.3 秒或者 300 毫秒。就像 .element 的值，也可以传递一个变量。我倾向于只在有多个元素和动画效果的时候使用变量的方式，比如想要使用同样的持续时间，但是这种情况比较少。

delay

```
TweenMax.to(".element", 2, { x: 100, delay:
    2 });
```

如果你想让动画在触发之前停留一段时间，可以使用 delay。delay 在锁住或者设置一个接着另一个的事件时很有用。在下一章中，我们还将介绍用时间轴工具来展示更多的效果和组织链式的效果。

动画的属性

我们简要地解释了示例代码中如何把球往右移动 100px，下面我们再解释一些相关的内

92 容。x 代表什么？它事实上代表 transform：translateX(100px)，就像 SVG 中的 rects，不应该对元素中的 x 属性感到困惑。记住，当我提到 transform 和 opacity 的时候，它们执行动画是最高效的。GreenSock 的开发者知道这些，所以他们很友好地创建了一些短命名，可以使用 x、y、z、scale 和 rotation（而不是 CSS 里面的 rotate）。可以单独使用而且可以在不同的时间使用，它们为我们节省了很多代码，而且让代码的可读性更高。

记住，如果你要让 SVG DOM 进行转换，它会使用 viewBox 的坐标系统，所以不要使用像素。你可能会想起之前的章节中有一个很有用的功能，我们可以简单地使用 scale 和创造复杂响应式的动画（在第 16 章会进行更多相关的介绍）

CSS 动画转换

因为 transform 是 CSS 的一个属性，所以在不同的时间对同一个元素应用不一样的 transform 会比较麻烦。它们最后采用一个层叠顺序并且一个接着一个地应用，除非每次变化的时候，你都设置每一个百分比的值。

我在 CSS-Tricks 发表了一篇文章（*http://bit.ly/2ivvcrP*），里面写了更多的内容。

GreenSock 用了很多个更新版本来改善这些属性的分离，这样我们才能更好地控制运动。

CSS 工作组也在计划把 transform 拆解成自己的属性，但是在实现的发布时间和浏览器的扩展上都不是很清晰。Chrome 有一些测试性的实现。

GreenSock 中同样有 opacity 属性，就像它在 CSS 里面的作用一样：支持 0 到 1 的值，0 代表透明，1 代表不透明。GreenSock 还提供了一个自定义的值，autoAlpha，同样也提供 0 到 1 的值。这个值连接着 opacity 和 visibility:hidden，这样就实现了移除元素或将元素添加到 DOM 中。

这个很重要，因为 opacity 设置为 0 的元素还是会和鼠标 / 触摸 / 键盘事件有联系，屏幕阅读器的可读性树上会包含它们。但是 visibility 设置为 hidden 的元素不会这样。autoAlpha 确保当元素完全退出时，就像在视图上一样，正常地隐藏起来，不受任何交互影响。

你也可以使用其他 CSS 的值来做动画，如颜色、宽度、高度还有透视，它们都可以。但是还是有一些事情需要记住。首先，任何一个属性的名字都需要变成驼峰式写法。举例如下：background-color 应写为 backgroundColor，border-radius 应写为 borderRadius。

93 同样，任何不是数字的值需要以字符串形式传递，并用引号包裹。所以，一个颜色的值

应该是："#333333"。

当使用两个属性来做动画的时候，我们用逗号将它们隔开（类似对象的属性）：

```
TweenMax.to(".element", 2, {
 x: 100,
 y: 50
});
```

easing

easing 是可选的，所以我没有在第一个示例中包含它。但是，easing 可能是 GreenSock 中最有用的工具，它给静态的代码带来了很多活力。可以像下面这样添加 easing：

```
TweenMax.to(".element", 2, {
 x: 100,
 y: 50,
 ease: Sine.easeOut
});
```

当我们这样写的时候，`ease:` 始终保持一致。这里，`Sine` 是 ease 的类型。大部分的 easing 曲线包含 3 个可选项：`.easeIn`、`.easeOut` 和 `.easeInOut`。它们倾向贝塞尔曲线的方式。还有很多 ease 的类型。在我之前学习的时候，我发现 GreenSock Ease Visualizer（*http://greensock.com/ease-visualizer*）是一个非常有用的工具（参见图 8-3），尤其是在观察和探索这些不同的可选项的时候。

最近，GreenSock 介绍了一个新的 ease 类型，名字叫 Custom。要使用它，你需要加载 CustomEase 插件，它可以传递 SVG 的路径，而且你可以用 Ease Visualizer 来操作这些路径（你可以观看示例的变化）。这是一个非常有用的功能，有时你采用的 easing 的类型可以让所有东西变得逼真而且给人留下深刻的印象。

94

easing 小技巧

95

尽管它们的名字可能给人提示，`.easeOut` 确实对进场很有用，`.easeIn` 对退场很有用，对于一些中间状态，我倾向于使用 `.easeInOut`。

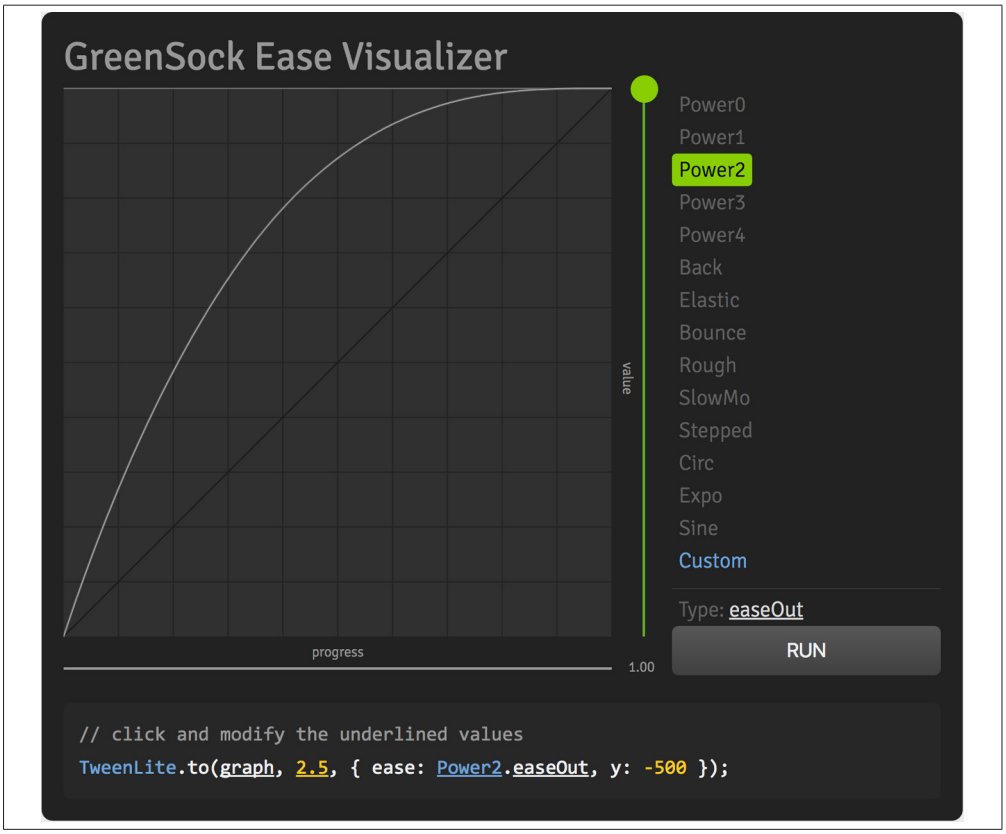


图8-3: GreenSock Ease Visualizer 是一个无价的交互式工具

这看起来有很多可以挖掘和领会的东西，但是在多次使用这些语法之后，就很容易记忆，因为你会一遍一遍地使用同样的形式。我强烈建议大家深究本章中的部分代码，这样你的印象会更深刻。

在后面的章节中，我们会挖掘更多高级和有趣的事情，现在，我们已经把基础内容介绍完了。

第 9 章 97

GreenSock中的时间轴

在第 8 章中，我们讲述了 GreenSock 中的 tweening 库函数。在这章中，我们将介绍 GreenSock 中我最喜爱的一个功能：时间轴（timeline）。

一个简单的时间轴

上一章，我们谈论了有关 GreenSock 的语法，重点集中在 `TweenMax` 基础部分。请大家回想一下，我之前提到过，TweenMax 包含 TimelineMax，TimelineMax 是 GreenSock 工具集中功能最全的时间轴控制工具。它优秀在哪里呢？

TimelineMax 是一个很强大的工具，它用于控制复杂动画和动画执行的先后顺序。 在使用它之前，需要先实例化一个 TimelineMax 实例，可以这样做：`var tl = new TimelineMax();`（当然，如果你使用的是 ES6 语法的话，也可以选择使用 `let` 或者 `const` 进行声明），虽然可以随便为变量取名，但是 `tl` 往往是行业内标准的声明方式。

你可以使用 TweenMax 做任何动画，但需要注意的是，TweenMax 默认会让你指定所有动画同时执行。如果想分步执行，那么需要给每个动画添加动画延时。

让我们来看看图 9-1 所示的三个图形动画，它们会一个接一个地横向从左向右运动。

到目前为止，根据我们已经学过的知识，代码可以这么写： 98

```
TweenMax.to(".star", 3, {x: 300, ease: Power4.easeOut});
TweenMax.to(".circle", 3, {x: 300, delay:3, ease: Power4.easeOut});
TweenMax.to(".hex", 3, {x: 300, delay:6, ease: Power4.easeOut});
```

以上代码虽然是能起作用的，但是如果场景一变，需要让更多的元素运动，就不得不一直手动计算每一个元素需要的延时，这样十分耗时耗力。所以这时候应该使用时间轴来

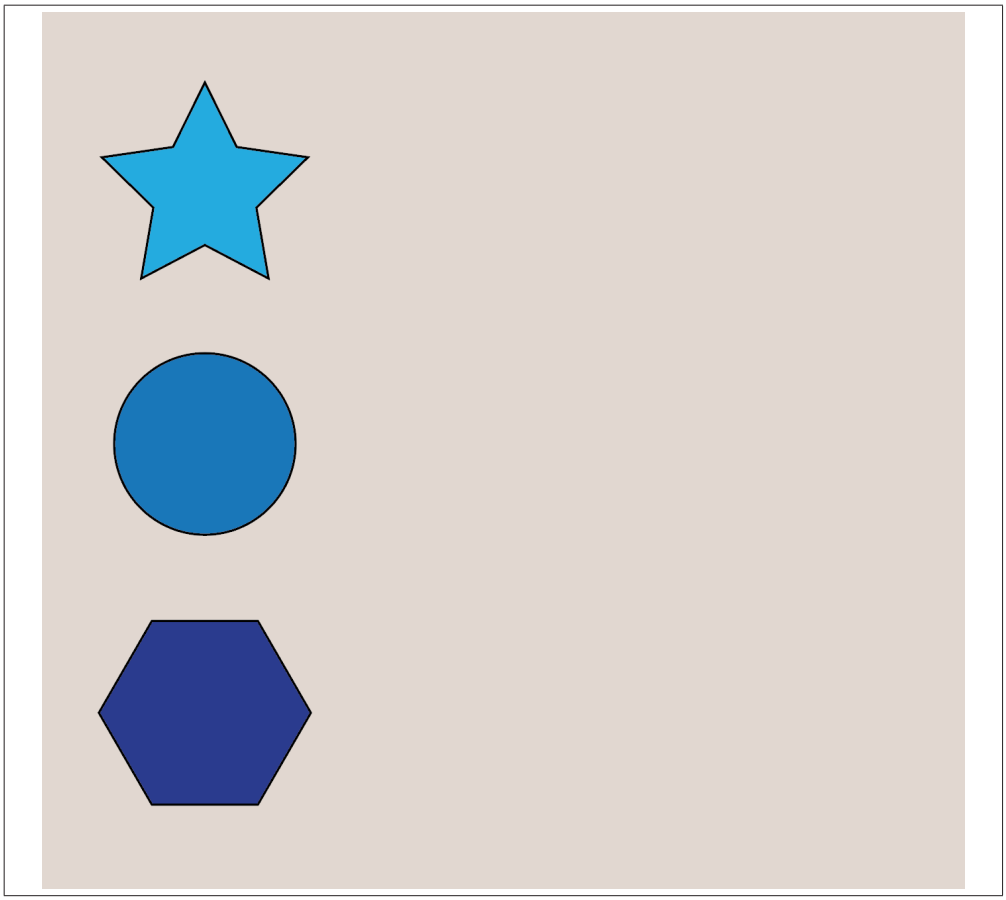


图 9-1：横向运动

维护这些动画，它能帮我们将目标元素一个接着一个自动地执行运动（注意，此时不需要计算和声明延时了）：

```
var tl = new TimelineMax();

tl.to(".star", 3, {x: 300, ease: Power4.easeOut});
tl.to(".circle", 3, {x: 300, ease: Power4.easeOut});
tl.to(".hex", 3, {x: 300, ease: Power4.easeOut});
```

99 这样的解决方案非常好，因为从动画的开始到结束，时间轴会依照我们指定的顺序自动执行每一个要执行的动画。

当我们看了这个动画觉得不太合适，决定调整圆的运动动画时，调整的方案是让圆的动

画在星星动画完成之前就开始。应该怎么设置呢？可以使用之前在 .staggerTo 动画中设置的位置参数，并使用相对的时间增量（相对前一个动画所设置的动画时长减 1 秒）：

```
var tl = new TimelineMax();

tl.to(".star", 3, {x: 300, ease: Power4.easeOut});
tl.to(".circle", 3, {x: 300, ease: Power4.easeOut}, "-=1");
tl.to(".hex", 3, {x: 300, ease: Power4.easeOut});
```

时间增量

在之前的例子中，我们使用了相对增量 "-=1"，它让时间轴了解我们想提前 1 秒执行下一个动画。如果你对 JavaScript 并不熟悉，增量对于你来说是很有用的。这种 +=1（或者任意的整型数）或 -=1 的语法，能够让编译器明白，我们希望在原先的基础上加上或减去 1 秒，而不是设置一个单纯静态的值。你可以看看下面多种用时间轴使用增量的例子。

例如，我们想在上一个动画完成后再加 1 秒的延时，可以这样写：

```
tl.to(".circle", 3, {x: 300}, "+=1");
```

如果想在上一个动画完成前 1 秒开始下一个动画，可以这样写（但是需要保证之前已经声明了上一个动画）：

```
tl.to(".circle", 3, {x: 300}, "-=1");
```

或者也可以设置一个特别的静态时间，例如 2 秒，那么动画就会准确地在第 2 秒开始执行：

```
tl.to(".circle", 3, {x: 300}, "2");
```

我最喜欢的是通过设置相对标签，这样可以对后面的所有动画都产生影响。

圆形在星星动画完成前 1 秒的时候开始运动，而后面的六边形则不会改变时间顺序，依然紧跟着圆形，当圆形动画完成后开始运动。我们在设计圆形的运动延时时，无须关心圆形之后的六边形的运动时间。（原来六边形开始运动的时间是在第 6 秒，而现在则在第 5 秒开始。）

相对标签

到目前为止，一切都很好，但是有一个问题，TweenMax 默认是串行执行的，如果需要完成一个非常复杂的动画，并且需要同时并行运行多个动画，或者你想轻缓地在某一个动画前后执行另一个动画，这里的动画逻辑会稍复杂和混乱。尤其是当你需要校准动画时间顺序时，相信我，在开发动画的过程中无时无刻都需要校准动画的时间顺序。这时

100

候就需要用到相对标签。相对标签十分好用，它可以作为一个时间基准点被插入时间轴中，或者在动画开始前和结束后。我们使用 `tl.add("相对标记名")` 这样的语法来声明一个相对标签。

让我们看一个简单的例子，如图 9-2 所示。

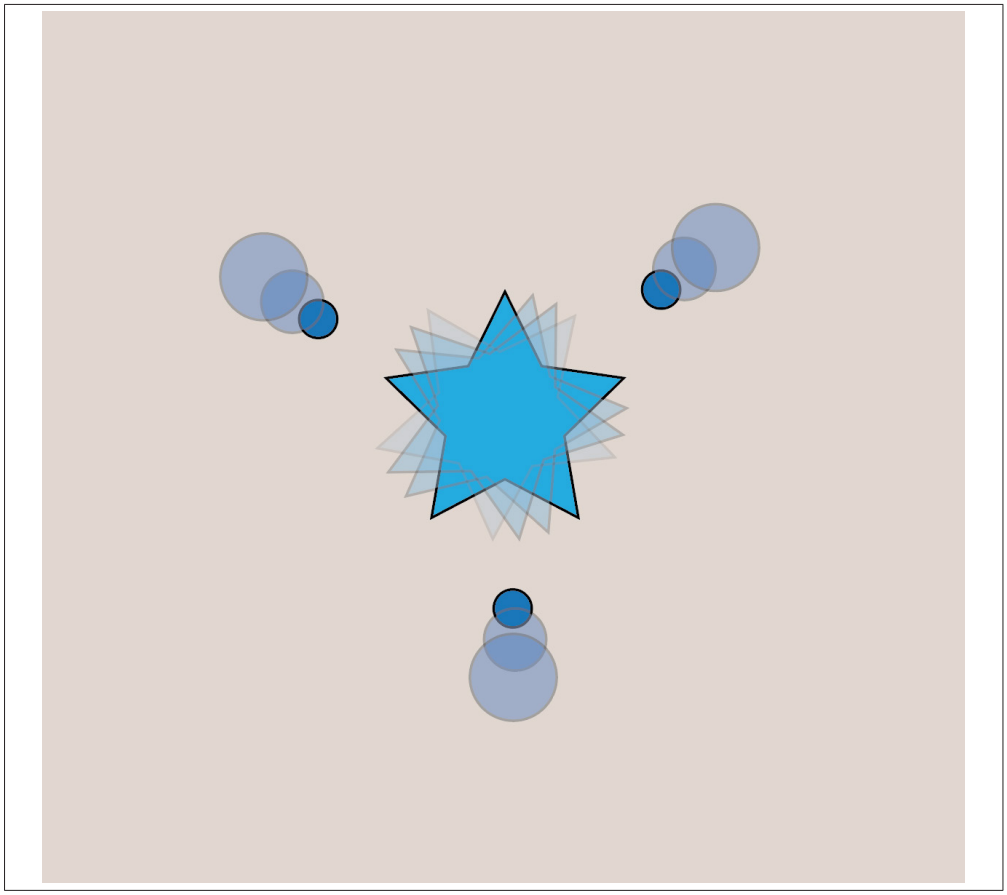


图9-2：这是一个星星自转和三个小球移动的动画

在这个例子中，我们可以看到星星在自转，当旋转完成后，三个小球同时放大并移向三个不同的方向。

虽然运动形式相同，但由于每一个小球的运动方向是不同的，所以我们不能对它们使用同样的动画形式（是的，我们可以创建一个函数进行复用，但是这要放在下一章进行介绍）。我们需要对它们逐个创建动画，而 `timelineMax` 默认是一个一个执行的，如果使用

原先的方式，需要像下面这样逐个计算并设置时间增量：

101

```
var tl = new TimelineMax();

tl.to(".star", 3, {
    rotation: 30,
    transformOrigin: "50% 50%"
});

tl.to(".circle1", 1, {
    scale: 2.5
    x: 100,
    y: -70
});

tl.to(".circle2", 1, {
    scale: 2.5
    y: -100
}, "-=1");  // 每次设置新动画，都需要基于第一个小球的时间点设置时间增量

tl.to(".circle3", 1, {
    scale: 2.5
    y: -70,
    x: -100
}, "-=2");
```

如果使用相对标签就会十分方便：

```
var tl = new TimelineMax();

tl.to(".star", 3, {
    rotation: 30,
    transformOrigin: "50% 50%"
});

tl.add("burst");  // 在第一个小球之前设置相对标记点

tl.to(".circle1", 1, {
    scale: 2.5
    x: 100,
    y: -70
}, "burst"); // 后续三个小球的运动时间都相对于标记设置的时间点进行运动

tl.to(".circle2", 1, {
```

```
    scale: 2.5
    y: -100
}, "burst");

tl.to(".circle3", 1, {
    scale: 2.5
    y: -70,
    x: -100
}, "burst");
```

相比于为每个小球加增量的做法，我更喜欢使用 `add("burst")`，因为这样看起来更加清晰灵活，所有的元素都会在时间轴中更早地运动，所有小球都会在同一时刻向三个方向运动。当设置了相对标签后，所有小球的运动时间同时改变了，不需要重新计算时间顺序。你甚至可以依据相对标签做出较晚或较早的时间调整。例如，如果想在时间点后一秒发射三个小球，只需结合之前学过的相对位置参数，像 `"brust+=1"` 这样，对相对标记点的时间设置增量就行了。

使用 .set 方法能使你的动画项目较稳定

你也许在让元素进行动画前很多次声明了某些属性，例如想对某个元素设置 z 轴方向的运动，你需要事先设置 perspective 才能看到效果，又或者，你为了使用 DrawSVG 动画库（*https://greensock.com/drawSVG*）进行描边动画，对一个图形的初始形态设置了无描边（`stroke-dasharray: 0`）。虽然这些事情可以在 CSS 中设置属性定义，但是更好的做法是使用 .set 函数在 JS 中设置元素的初始状态，因为对于那些读你代码准备进行维护的其他人来说，可以明确知道你接下来需要对哪一块做动画。而如果设置在 CSS 内、JavaScript 之外，其他人就不容易理解运动的初始状态是什么（还要去查对应的 CSS 文件），所以使用 .set 设置元素的初始状态，是利于他人后期维护的。总的来说，可以用 tl.set 方法设置状态且这种方式不会对元素造成任何移动：

```
tl.set(".circle", {scale: 0.5});
```

也不需要总是使用 .set 设置，如果你设置的是 .fromTo 动画，可以直接指定元素从初始状态变化到另一个状态，直接完成整个动画。让我们来看看应该怎么做：

```
tl.fromTo(".circle", 1, { // 原书缺少 repeat 参数，不加上会报错
    scale: 0.5
}, {
```

```
        scale: 1
    }, 8);
```

虽然这个动画设置了 8 秒延迟，但是一开始的初始状态会保持到整个动画开始之前。

即便如此，在某些场景中，即使你使用了 .from 或者 .fromTo，动画也还是需要 .set 函数来辅助的。例如有这样一个场景：一开始你设置了某个元素透明度的值为 0，你可能会注意到在 JavaScript 还在加载的过程中有一瞬间元素会显现出来，这是由于 JavaScript 还没有加载完，没来得及对元素设置透明所导致的，虽然这通常只有一瞬间，但还是能被肉眼察觉到的。这种情况该怎么解决呢？我们可以这样做：由于 CSS 是首先加载的，所以在 CSS 中进行设置：

```
.element { visibility: hidden; }
```

这样元素一开始是被隐藏的。然后在 JavaScript 中使用 .set 处理，将隐藏的元素显示出来，并且设置淡入动画：

```
TweenMax.set(".element", {visibility:"visible"});
```

这一招，我在每个动画项目中都屡试不爽。

主时间轴和所嵌套的场景

103

我不建议将动画声明函数写在 JS 的全局域中。如果是一个比较简单的动画，可以把动画逻辑放在 IIFE（immediately invoked function expression，自执行函数表达式）中；如果是有点规模的动画项目，应该通过函数形式进行封装并调用。

而更进一步的方式则是先创建一个主时间轴。我之所以喜欢这样做，是因为可以把整个动画切分成很多个场景，这样便于更好地进行控制，例如：

- 当需要对某个场景动画进行调整的时候，可以根据场景对应的名字和场次快速定位到需要微调的位置。
- 方便对整个场景进行定位。
- 每次只做一个场景，效率更高。
- 可以使用 .seek 方法调整某个场景的运动时间，这样不用在整个场景动画中一帧一帧地去调整。
- 只需为主时间轴绑定一个事件就能统一控制重播和播放，例如单击事件。
- 保持整个动画代码是有组织、有逻辑，且整洁易读的。

让我们继续深挖，来看看如何创建一个主时间轴，我将会解释每一个精华的部分。

代码的逻辑组织

来看看这个例子，如何设置一个子时间轴，并将其嵌套进主时间轴。

```
function sceneOne() {
    var tl = new TimelineMax();

    tl.add("begin");
    tl.to(".bubble", 2, {
        scale: 3,
        opacity: 0.5,
        rotation: 90,
        ease: Circ.easeOut
    }, "begin");
    ...

    return tl;
}

var master = new TimelineMax();
master.add(sceneOne(), "scene1");
```

104

在这个例子中，我们定义了一个函数，并且在函数中声明了一个时间轴实例并设置了一系列动画（这个函数也不一定要叫 `sceneOne`——可以给它命名为任何你想要的名字）。在函数的最后，返回这个时间轴实例，然后把这个场景动画函数添加到主时间轴中。你也许注意到了，我为这个场景设置了一个定位标签—— `scene1`，这样做可以让我在工作中通过这个标签来控制 sceneOne 这个场景。

现在，让我们尝试在主时间轴中插入更多的场景，这些都是没有技术含量的工作：

```
function sceneOne() {
    var tl = new TimelineMax();

    tl.add("begin");
    tl.to(".bubble", 2, {
        scale: 3,
        opacity: 0.5,
        rotation: 90,
        ease: Circ.easeOut
    }, "begin");
    ...
```

```
    return tl;
}

function sceneTwo() {
    var tl = new TimelineMax();

    tl.add("boom");
    tl.to(".star", 2, {
        scale: 5,
        opacity: 0,
        rotation: -360,
        ease: Circ.easeIn
    }, "boom");
    ...

    return tl;
}

var master = new TimelineMax();
master.add(sceneOne(), "scene1")
      .add(sceneTwo(), "scene2");
```

也可以根据自己的想法来简单改变每一个场景的顺序：

```
var master = new TimelineMax();
master.add(sceneTwo(), "scene2")
      .add(sceneOne(), "scene1");
```

使用 .seek 方法实现更好的工作体验

105

你写的动画终归会变得越来越长。这时候对于部分单一动画的调整会变得比较困难。你必须时刻提醒自己注意正在修改哪些东西。为了解决这个痛点，我们在工作中需要使用 .seek 函数。这时候，我们前面声明的位置标签就变得十分有用。有时，当调整一个只有几秒的小动画时，可能会突然忘记自己想做什么。这时就可以用 .seek 函数，把多个场景分离开，再调整单独的部分场景：

```
var master = new TimelineMax();
master.add(sceneOne(), "scene1")
      .add(sceneTwo(), "scene2");

master.seek("scene2+=3"); // 让当前调整的场景整个加 3 秒，
                          // 从而使其和其他场景分离开
```

当完成了一段很长的动画时，我们经常会注意到有一段场景有少许快或少许慢。由于动

画较长，不方便再微调，这时候就可以发挥 timeScale() 方法的优势了。这是 GreenSock 中一个十分有用的功能，你一定会因为它提供的便利感到很惊奇。我会将一段长动画分成不同的组织结构，然后使用 timeScale() 来对这些结构进行微调，来看看下面这个例子：

```
function sceneOne() {
    var tl = new TimelineMax();

    tl.add("begin");
    tl.to(".bubble", 2, {
        scale: 3,
        opacity: 0.5,
        rotation: 90,
        ease: Circ.easeOut
    }, "begin");
    ...

    tl.timeScale(1.5);

    return tl;
}
```

使用 timeScale() 使场景 1 中的所有动画快 1.5 倍，如果想调慢动画，那么可以设置 timeScale() 小于 1（例如 0.5），这样所有的动画将慢 50%。

循环

如果想让一个动画不断地循环执行，可以设置其 repeat 属性为 -1，但是如果只想让单独的某一个场景无限循环呢？可以在该场景的时间轴上加一个参数。来看看下面这个例子：

106

```
function sceneOne() {
    var tl = new TimelineMax({ repeat: -1 });

    tl.add("begin");
    tl.to(".bubble", 2, {
        scale: 3,
        opacity: 0.5,
        rotation: 90,
        ease: Circ.easeOut
    }, "begin");
    ...

    return tl;
}
```

如果想让动画像悠悠球一样在初始状态和最终状态之间来回循环，可以用 yoyo 功能，语法是设置一个布尔值 yoyo: true。yoyo 功能可以应用于单个补间动画、整个子时间轴，或者整个主时间轴。但是只有时间轴被设置为循环播放的时候，yoyo 才会起作用。如果想给整个主时间轴应用 yoyo 功能，需要在主时间轴上使用下面的方式：

```
var master = new TimelineMax({repeat: -1, yoyo:true});
```

当某一个动画反复运行的时候，我们就会意识到声明一个明确的延时对于动画来说是多么有益。delay:1 会将某一个时间轴上的所有动画都延迟 1 秒，但整个动画重新执行的时候不会暂停。而设置 repeatDelay: 1 则是将整个动画的执行延迟 1 秒，而不对时间轴上的某个动画进行延时。

重复用法

接下来讲到的方法对于 TimelineLite 和 TimelineMax 都适用，但是循环、反复或者 yoyo 反复都只对 TimelineMax 起作用。

如果一段动画是循环或者重新被触发的，你需要考虑将动画内所有的 .to 方法都替换为 .fromTo 方法。这样可保证每一次动画执行的时候都是从初始状态到结束状态。

不仅可以通过设置 repeat:-1 的方式来使动画循环，还可以通过设置 onComplete 回调函数的形式来触发。如果想将动画设置得更加精巧，回调函数会给你带来很多帮助，让我们来看看用回调方式让动画循环的例子：

```
function _flyBy(el, amt) {
    TweenMax.to(el, amt, {
        x: -200,
        rotation: 360,
        onComplete: this._flyBy,
        onCompleteParams: [el, amt]
    });
}
```

107

如果想通过回调函数来设置一些随机的效果，可以这样做：

```
function _flyBy(el) {
    TweenMax.to(el, amt, {
        x: Math.random() * 400 - 200,
        rotation: Math.random() * 360,
        onComplete: _flyBy,
```

```
        onCompleteParams: [el]
    });
}
```

onComplete 允许我们设置一个回调函数（当动画已经完成的时候就会执行），而 onCompleteParams 参数则允许我们以数组的形式给 onComplete 的回调函数传递参数。通常来说，如果只是循环一个随机数，编译器会先产生一次随机数，然后不断重复使用这个随机数，这样就不算是真的每次都随机了。而在上面的代码中，我们让每次动画一结束就重新执行 _flyBy 函数，且都带上 el、amt 和可变的参数（编译器每一次都会产生一个新的随机数）。 所以这是 TimelineMax 编译器较好的地方。

下面列出了很多回调函数，供我们在不同的动画生命周期中使用，其中有绑定回调函数作用域的回调函数、当所有回调执行完后的回调函数等。

这些可用的回调函数是：

- onStart
- onStartScope
- onStartParams
- onComplete
- onCompleteScope
- onCompleteParams
- onUpdate
- onUpdateScope
- onUpdateParams
- onRepeat
- onRepeatScope
- onRepeatParams
- onReverseComplete
- onReverseCompleteScope

callbackScope 可以为对应的回调函数绑定合适的上下文，其中包括 onStartScope、onUpdateScope、onCompleteScope、onReverseCompleteScope 和 onRepeatScope 这几种方法。

108 暂停和暂停事件

你也许还记得，我们之前提到的通过在 TimelineMax 中以对象的形式设置 repeat:-1，

可以让整个时间轴结束后再次重新开始。同样的，你也可以设置整个时间轴在初始状态下暂时停止：

```
var master = new TimelineMax({paused: true});
```

如果想让时间轴在页面加载的时候暂时停止，在单击动画时才会执行，可以这样写：

```
var master = new TimelineMax({paused: true});
...

var el = document.getElementById("button")
    el.addEventListener('click', function(e) {
    e.preventDefault();
    master.restart();
}, false);
```

非常简单!

当然，我们可以在这里列出所有事件，但没什么意义。借助一些工具，例如 Hammer.js（用于在 PC 端实现手机端手势的 JS 框架库，*http://hammerjs.github.io/*），你就可以使用移动端的手势来触发动画了，就好像在原生移动端的滑动、双击等。

在第 12 章，我会给大家讲述如何使用 GreenSock 的时间轴的拖曳功能调节动画。我们还可以使用 jQuery 提供的 slider UI 来做一个对于控制时间轴的简洁界面的控制接口。Chris Gannon 的 ScrubGSAPTimeline（*http://bit.ly/2lYp1P4*）工具可以说是控制时间轴最棒的工具了。这里我写了一个例子告诉大家如何使用它（*http://bit.ly/2mB4pKv*）。

其他关于时间轴的方法

GreenSock 的时间轴工具提供了太多的功能，以至于一页纸都罗列不完，以下罗列的只是一部分你可能会用到的功能。我们之前介绍了我认为最基础的一些功能，剩下的很多有趣的功能需要你自己去阅读 GreenSock 的文档（*https://greensock.com/docs/#/HTML5/*）或者在 GreenSock 论坛（*https://greensock.com/forums/*）中和网友们讨论。许多方法的名字看起来好像很直观，但实际上还有很多妙用，文档会帮助你去更深入更清楚地理解它们的用法。

请注意，由于有很多 ActionScript 版本的 GSAP（官网已经不维护 AS 版本的 GSAP 了），它们看起来实在是太过老旧了，本书使用统一的 JS 版本的 GSAP 进行介绍。

109 TimelineLite 和 TimelineMax 通用的方法

- add()
- addLabel()
- addPause()
- call()
- clear()
- delay()
- duration()
- eventCallback
- exportRoot()
- from()
- fromTo()
- getChildren()
- getLabelTime()
- getTweensOf()
- invalidate()
- isActive()
- kill()
- pause()
- paused()
- play()
- progress()
- remove()
- removeLabel()
- render()
- restart()
- resume()
- reverse()
- reversed()
- seek()
- set()
- shiftChildren()
- staggerFrom()
- staggerFromTo()
- staggerTo()

- startTime()
- time()
- timeScale()
- to()
- totalDuration()
- totalProgress()
- totalTime()
- useFrames()

110

只存在于 TimelineMax 中的方法

- currentLabel()
- getActive()
- getLabelAfter()
- getLabelBefore()
- getlLabelsArray()
- repeat()
- repeatDelay()
- tweenFromTo()
- tweenTo()
- yoyo()

掌握了时间轴的基础知识，下面我们就开始制作一些真正有趣且富含想象力的动画吧。

111

第 10 章

MorphSVG 和路径动画

GreenSock 的插件有许多让人惊奇的特性。首先我们来说说 MorphSVG 和路径动画（BezierPlugin），这两个特性对于实际运动的模拟度很高。

加载插件

MorphSVG 是一个值得投入的插件，它可以在你还没为它付钱之前先试一试。GreenSock 提供了这些插件的 CodePen-safe 版本，这样你可以直接在 CodePen（*http://bit.ly/2lv1xhg*）里面试一试。

不要忘记在开始动画制作之前引入插件的 JS 脚本和 *TweenMax.min.js*。

MorphSVG

MorphSVG 是 GreenSock 中最令人激动的一个特性。起初，GreenSock 仅仅是一个支持通过不同路径点进行渐变运动的库。SnapSVG、SMIL，甚至是 D3 都可以将一个路径图形变为另外一个，但是如果这些路径点是不规则的，这个变形可能会失败或者看起来非常奇怪。而 MorphSVG 可以在不规则路径下让图形更流畅地变形，并且通过 `findShapeIndex()` 方法完美地调整创造出来的 Morph 类型。

将一个 SVG 路径变为另外一个，只需要将它的 ID 改为另外一个即可。没开玩笑，就是这么简单。然后你就可以创造出让人惊叹的效果了。语法如下：

```
TweenMax.to("#pathFrom", 1, {morphSVG:"#pathTo"});
```

112 MorphSVG 可确保路径在视窗中的位置，所以需要注意，在制作 SVG 时，应该让变形的两个路径在同一个位置，或者在开始动画之前将它移动到其他位置上。

还可以通过之前介绍的 ID 语法，或者直接以字符串的形式提供多边形的点，来渐变多边形线条或者多边形元素：

```
TweenLite.to("#polygon", 2, {morphSVG:"10,10 40,70 70,70 70,10"});
```

MorphSVG 是专门为路径和多边形 / 线条设计的，但是有时候可能需要变换圆、矩形或者其他 SVG 元素。而这个插件提供的 `convertToPath()` 方法很容易完成这个要求。既可以通过与目标元素关联的 ID 或者类名来调用，也可以直接传入完整的元素一次性从开始变形：

```
MorphSVGPlugin.convertToPath("circle, rect, ellipse, line, polygon, polyline");
```

插件的兼容性

MorphSVG 和 TweenMax 都很复杂，并且它们接受定期更新，所以使用兼容版本的文件很重要。我曾看到过很多人因为使用了最新的 TweenMax 版本和较老的 MorphSVG 版本制作动画而失败的案例，所以如果你在制作一个简单的变形动画时遇到问题（我也会先从简单的开始），请先检查一下你使用的是否是兼容的版本。

findShapeIndex()

MorphSVG 在通过不同路径点计算哪种过渡动画看起来更合适时，表现出很棒的计算性能，所以，使用 `shapeIndex` 属性的默认值 `auto` 就足够了。不过，偶尔想调整一下两点间的运动也没问题。

如果你特别关注形状变形的方式的话，`findShapeIndex()` 工具函数插件通过不断在循环点之间变换，可让你选取针对你的动画最合适的变形类型。先加载它，然后将元素的 ID 变为另外一个（比如，`findShapeIndex("#hex", "#star");`），随后一个很棒的 GUI 就出来了。不过，不要将上述代码遗留在你的代码库里，可以使用它，但在生产版本中删掉，这样可以减少不必要的冗余代码。

在图 10-1 和对应的 demo（*http://bit.ly/2g8CCyg*）里，可以看到我是如果将一个五角星变为十边形的，但是，`index` 可以完全改变它们插入的方式。路径点越多，你的选择就越复杂，更少的点提供更少的选项。

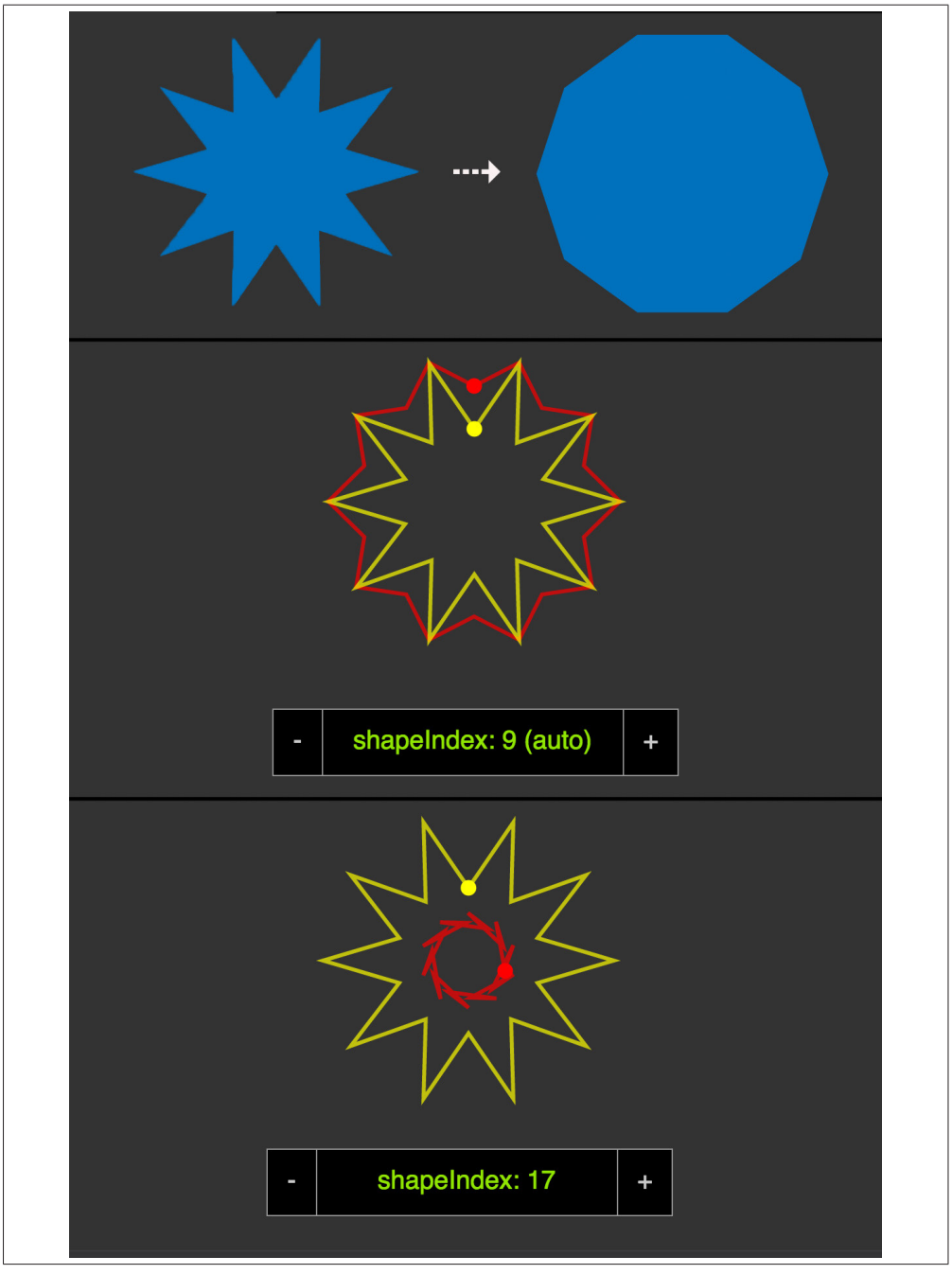


图10-1：我们打算变形的两个图形（顶部），当 ShapeIndex() 加载完成时，GUI 就会出现并且 shapeIndex 会被设置为 auto（中间），当将 index 设置为其他整数时，就会发生不同的形状变形

路径动画

路径动画对于动画里的实际运动来说是非常重要的。在 *x*、*y*、*z* 方向上插入一些单个值的方式非常受限。我们知道，一个瓶子中的萤火虫或者其他生物很少沿着直线飞行。现在，CSS 还不支持沿着路径运动的动画，不过已经被列为提案，你可以在 Microsoft Edge 上去投票支持其实现。SMIL 可以提供路径动画，但是 IE 或者 Edge 都不支持。

GreenSock 通过 BezierPlugin 插件，提供了一个很方便的方式创建上述效果。TweenMax 也支持相同的效果，而且它的兼容性为 IE 8+（对于 HTML 内容，从 IE 9 开始支持 SVG）。所以，路径动画现在差不多被浏览器全部支持了，并且还有回退兼容的方案。

为了创建一个路径动画，需要给 `bezier` 属性传入一个坐标数组：

```
TweenMax.to($firefly1, 6, {
    bezier: {
        type: "soft",
        values:[{x:10, y:30}, {x:-30, y:20}, {x:-40, y:10},
                {x:30, y:20}, {x:10, y:30}],
        autoRotate: true
    },
    ease: Linear.easeNone,
    repeat: -1
}, "start+=3");
```

我通常使用 *x*、*y* 坐标的值，因为它是我们之前提到的变换坐标。不过，其他值也可以正常使用，比如，`left`、`top`，甚至是 `rotation`。这意味着你可以通过添加 *z* 轴的值，沿着 3D Bézier 路径进行变化。这看起来确实让人很感兴趣，不过 99% 的时间，我都只使用 *x* 轴、*y* 轴的值。

当使用 *x* 轴、*y* 轴的值时，点的坐标是相对于元素的位置而不是 canvas 本身。换句话说，如果你定义了 `x:5,y:10`，那么这个运动会相对于距元素右边 5 和元素下部 10 的距离开始。后续的点同样都相对于元素的起始位置，并不是前一个变换位置。这可以让区域中的绘制点更容易围绕元素映射。在上述奇特的萤火虫飞行动画里（*http://codepen.io/sdras/full/MYQxXe/*），为了让其在灯泡的范围内运动，我扭曲了它的运动路径，这可能看起来有点轻微的跳跃，就像在实际运动中，它类似绕着一点运动（参见图 10-2）。

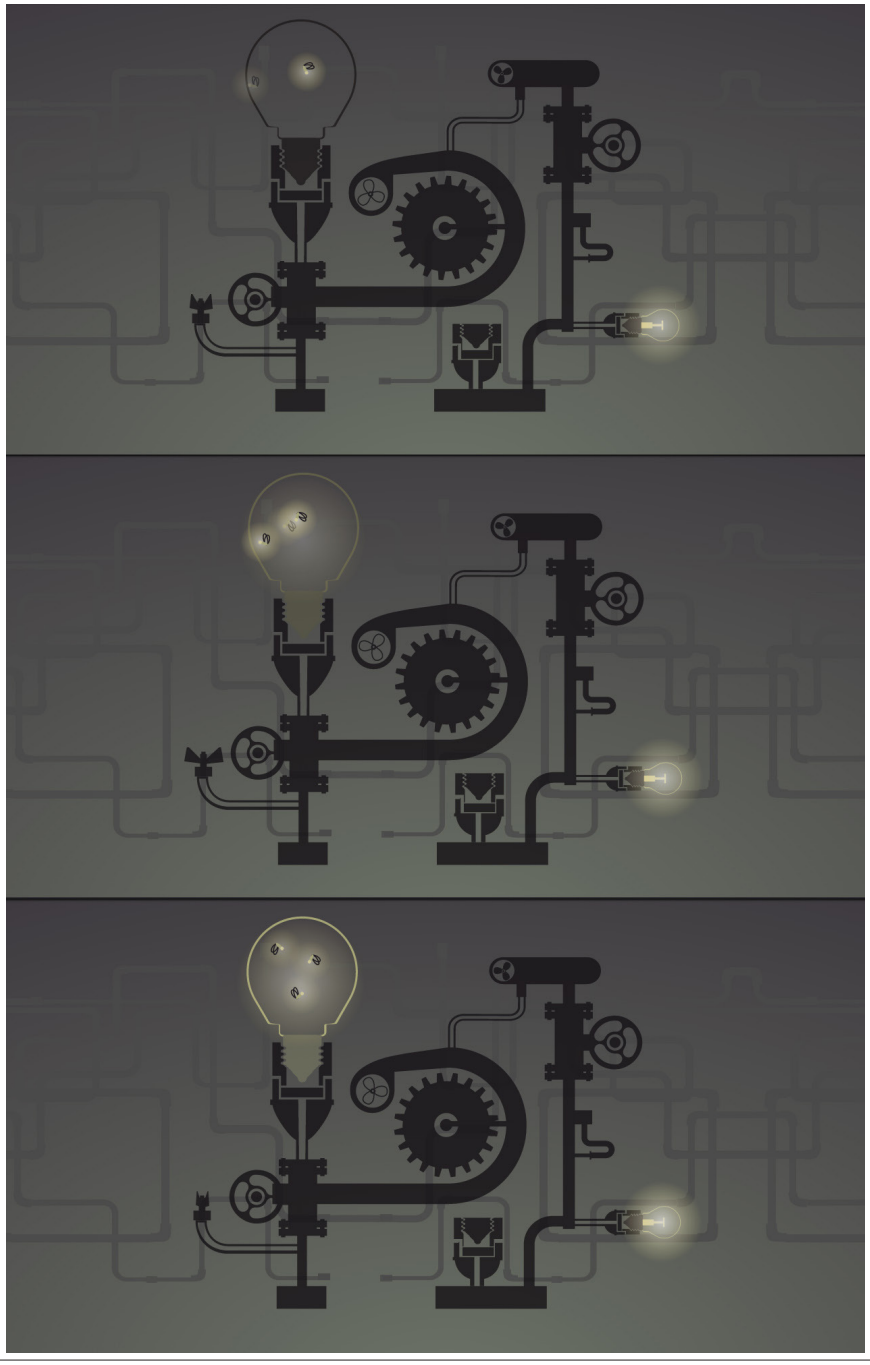


图 10-2：萤火虫以奇特的规则运动。为了让其更接近真实的运动，它们不应该以线性的方式运动而是应该绕着某条路径旋转运动

你可能并不想让萤火虫运动，只是想使用路径作为一个常规的坐标系，而且需要路径之 116
间更加平滑和精确。这里有两种办法可以做到。第一种是将 type 参数设置为 soft。这会使你提供的路径和曲线靠近这些点，好像它们朝着点的方向被拉升一样，而不是仅仅通过在两者中间插一个值，然后接着到下一个。第二种是将 type 参数设置为 thru（这是默认值），然后定义 curviness 属性值。0 表示无曲率，1 表示正常曲率，2 表示两倍曲率，依此类推。图 10-3 和对应的例子（*http://codepen.io/sdras/full/PqEPqz*）展示了不同设置的效果。

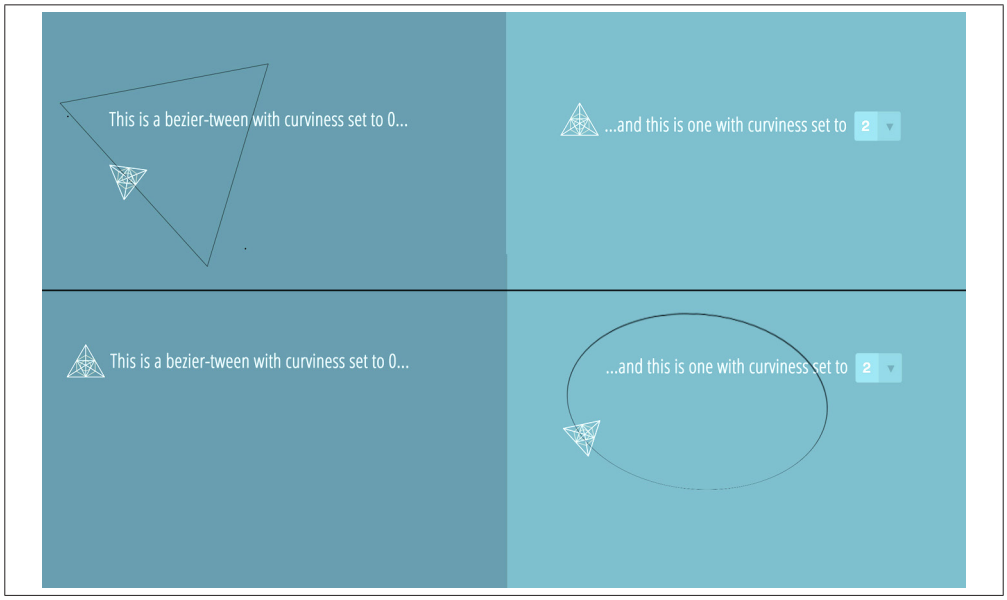


图10-3: 例子展示了不同 curviness 属性值的曲线运动

注意，当你传入值 3 时，曲线看起来有点不平滑，这是因为每一个点开始绕着它自己的轴循环。你可以将这个运动形式比作一个正在拉伸的橡皮筋：当 curviness 的值为 0 时，橡皮筋是被拉紧的；当设置为 2 时，橡皮筋的松紧刚好可以让不同点间的运动看起来平滑。当值达到 8 以上时，这个路径就没什么效果了。

除了 thru 和 soft 值外，还有两种 Bézier 类型的定义：quadratic 和 cubic。quadratic 允许在每个锚点处定义一个控制点。cubic 与之类似，不过你可以在每个锚点处定义两个控制点。对于 quadratic 和 cubic，必须以锚点作为数组的开始和结尾，尽管你可以在其中放入很多其他的值。

现在，你可以传递一个坐标系数组，虽然我并不惊讶在未来 GSAP 会提供使用 SVG 路 117
径本身来作为运动的定义。这个库一直在增添新的功能，你可以看看这个库（*https://*

github.com/greensock/GreenSock-JS/）的更新以及它在过去一年有哪些新加入的东西。

这里我同样要提到旋转。在前面的 pen 示例中，我简单地设置 `autoRotate: true`，在遍历数值数组时，使萤火虫绕着对应直线方向的轴进行旋转。你可以更精确地将 `autoRotate` 设置为一个整数而不是布尔值，使元素在初始状态就具有一定的旋转角度。或者，你也可以传入一个数组，去调整这些选项：

1. 第一个位置（为 `x`）。
2. 第二个位置（为 `y`）。
3. 旋转属性（通常为 `rotation`，假如你想使其在一个轴上的话，也可以写为 `rotationX` 或者 `rotationY`）。
4. （可选）度数（或者弧度），设置额外的旋转角度。
5. Boolean 值表示旋转属性是使用弧度单位还是角度单位（默认为 `false`，表示使用角度单位）。

将 `autoRotate` 设置为一个数组 `["x","y","rotation",0,false]`，这和设置 `autoRotate:true` 是一样的效果，并且该元素会沿着旋转路径进行运动。这 5 个参数中我最常使用的是第 4 个：该数值表示的度数会被加到新的旋转角上。这可以让你在动画设置之外，使字符或者元素在一定的方向上倾斜。这个参数用处很大，因为在实际运动一开始，会稍微调整一下倾斜方向，它们并不会仅仅以一个稳定的水平方向进行运动。

我提供了一个切换 `autoRotate: true` 和 `false` 的对应例子（*http://codepen.io/sdras/full/aOZOwj/*），可参见图 10-4 和图 10-5，你可以清楚地看到不同参数对于动画的影响。

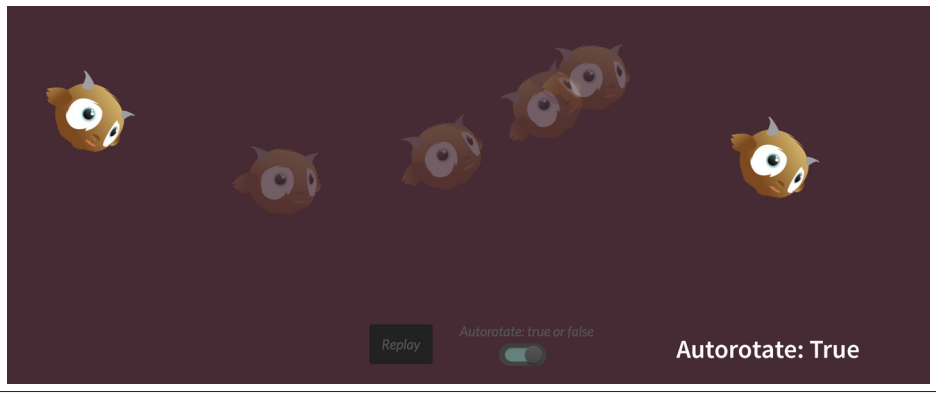


图10-4：当autorotate：true 时，元素/字符会沿着路径进行倾斜

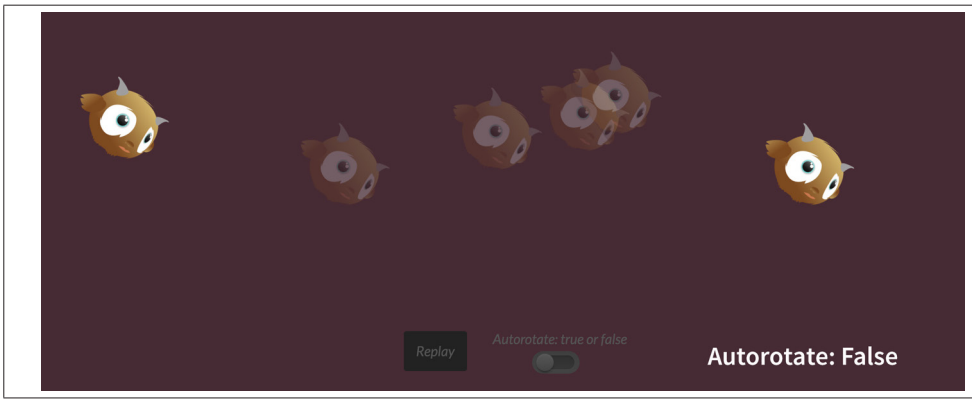


图10-5：当 autorotate：false 时，元素/字符不会沿着路径进行倾斜，这看起来有点奇怪

下面是具体的代码：

```
function lilGuyGo(autoRotate) {
    // 回到开始位置，并清除所有补间动画
    tl.progress(0).clear()
    // 将初始的旋转值设定为接近它的方向
    .set(lilG, {
        rotation: 40
    });
    // 在补间动画的时间轴上添加 Bezier 路径
    tl.to(lilG, 3, {
        bezier: {
            type: "soft",
            values: [
                {x: 0, y: 50}, {x: 150, y: 100}, {x: 300, y: 50},
                {x: 500, y: 200}, {x: 700, y: 100}, {x: 900, y: 80}
            ],
            autoRotate: "true"
        },
        // 添加缓动函数
        ease: Circ.easeInOut
    });
  }
lilGuyGo(true);
```

以这种方式运动，这个小物体比起给它设置一个固定角度的方式运动看起来更生动。你还可以看到，在开始的时候，我将它的脸沿着路径朝下，这是因为，如果我不这样做，它就会有一点“跳跃”。也可以通过传入一个数组作为 autoRotate 的一个选项的方式，像之前说的。这两者均可。

当然，路径运动并不仅仅局限于物体运动。当结合透明度或者变形动画时，会有无穷的可能性展现在动画上。

121 第 11 章

交错效果、Tweening HSL和SplitText的文本动画

交错的动画

很多 JavaScript 动画库都提供了交错动画的特性，这个特性非常善于创造酷炫的动画效果，与直接使用 CSS 创建相比优势巨大。如图 11-1 所示，让我们看一下编写动画的多种方式。

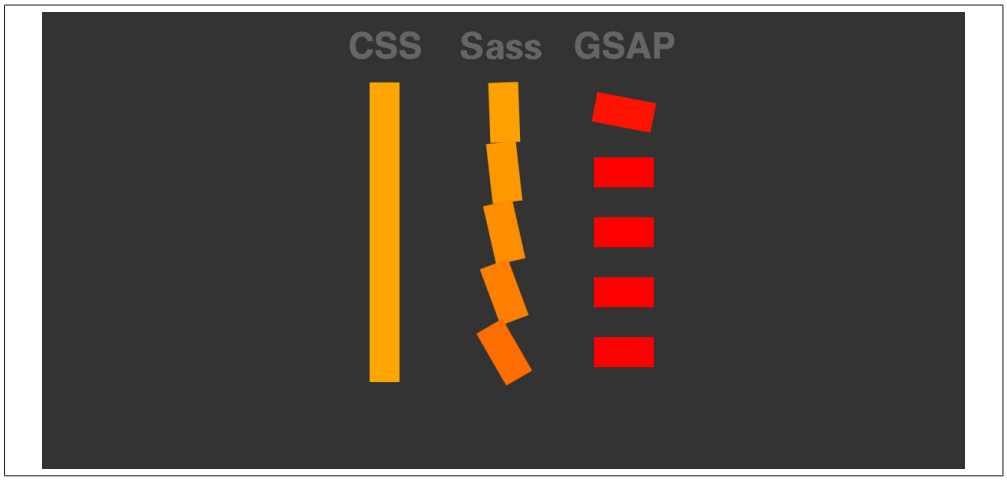


图11-1：使用CSS、Sass与GSAP编写相同的交错动画效果的对比

在 CSS 中创建交错动画，需要在元素或元素的伪类上使用相同的关键帧，并对每个关键

帧依次增加延时：

```
@keyframes staggerFoo {                                          122
    to {
        background: orange;
        transform: rotate(90deg);
    }
}
.css .bar:nth-child(1) {
    animation: staggerFoo 1s 0.1s ease-out both;
}
.css .bar:nth-child(2) {
    animation: staggerFoo 1s 0.2s ease-out both;
}
.css .bar:nth-child(3) {
    animation: staggerFoo 1s 0.3s ease-out both;
}
.css .bar:nth-child(4) {
    animation: staggerFoo 1s 0.4s ease-out both;
}
.css .bar:nth-child(5) {
    animation: staggerFoo 1s 0.5s ease-out both;
}
.css .bar:nth-child(6) {
    animation: staggerFoo 1s 0.5s ease-out both;
}
```

在 Sass 中，可以把以上代码精简一下：

```
@keyframes staggerFoo {
    to {
        background: orange;
        transform: rotate(90deg);
    }
}
@for $i from 1 through 6 {
    .sass .bar:nth-child(#{$i} ) {
        animation: staggerFoo 1s ($i * 0.1s) ease-out both;
    }
}
```

不过使用 GSAP 的话，只用一行代码就可以完成这个效果：

```
TweenMax.staggerTo(".gsap .bar", 1, {
    backgroundColor: "orange",
    rotation: 90,
```

```
    ease: Sine.easeOut
}, 0.1);
```

事实上，这种简洁的方式非常适合动画制作流程，特别是需要经常调整动画效果时。

使用 cycle 属性可以传递多个需要在其中交错的值，从而代替很多本来需要 Sass 来生成的、复杂的 nth-child 选择器。cycle 方法接受一组数组作为参数，并将这组数之间的值应用到元素上：

```
TweenMax.staggerTo(".foo", 1, {
    cycle: {
        y: [75, 0, -75]
    },
    ease: Power4.easeInOut
}, 0.05);
```

123 这些参数也可以随机化，从而制造出更多有趣的效果。这种方式优于创建一个随机函数并不断调用，因为这样写只会执行随机函数一次（因此之后不会随机）。这样使用 cycle 属性可以更容易地创建华丽且有趣的效果。下面的代码就使用了 cycle 属性来给每个动画随机生成值：

```
var coord = [40, 800, 70, -200];

TweenMax.staggerTo(".foo", 1, {
    cycle: {
        x: function(i) {
            return coord[Math.floor(Math.random() * coord.length)];
        }
    },
    ease: Power4.easeInOut
}, 0.1);
```

了解动画中的 Math.random()

Math.random() 能够创建迷人的随机效果并生成动态代码，因此对于 JavaScript 动画来讲它是非常有用的。Math.random() 返回一个 0 到 1 之间的值，因此对于一些场景的特性比如透明度，这个值可以开箱即用。在其他情况下，可以倍乘这个值直到其达到需求。在前面的代码中你可能已经注意到，Math.random() 方法放在了 Math.floor() 方法中。这样做的结果是，Math.random() 产生的值会向下取整到最小的整数值（Math.ceil() 向上取整，Math.round() 四舍五入取整）。

在这个场景下 .floor() 与 .round() 均适合，而我通常选择 .floor()，因为 .floor() 的性能略好。当然，如果不需要整数值，可以不用取整。

如果需要使用 Math.random() 来获取一个区间上的随机数，可以倍乘这个区间，并加上最小值，如下所示：

```
Math.random() * (max - min) + min;
```

在图 11-2 以及相关例子中（*http://codepen.io/sdras/pen/XmmjQb*），我让每个目标元素在三个值之间交错产生动画，只使用了一点点代码（22 行 JavaScript 代码），你也可以完成！

124

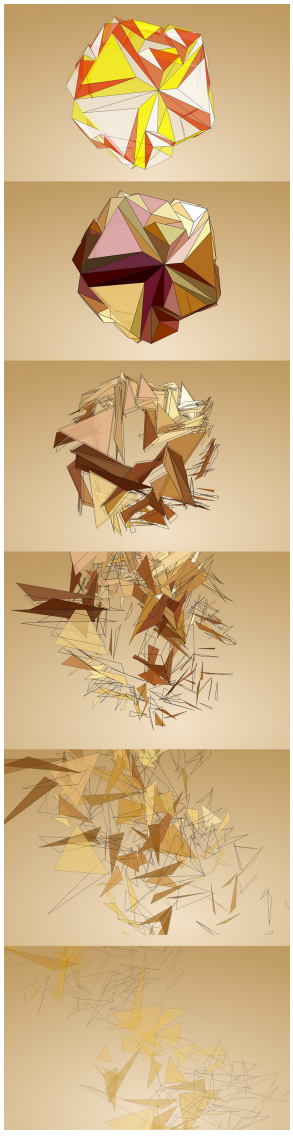


图11-2：所有的动画都依靠GSAP的补间动画能力，且只用了很少的代码

125 下面是具体的代码：

```
var bP = $(".boggle path"),
    tl = new TimelineLite();

tl.add("start");
tl.staggerFrom(bP, 3, {
    cycle:{
        fill:["white", "yellow", "#e23e0c"],
        opacity:[0.8, 0.2, 0.5, 0.3],
    },
    ease:Elastic.easeOut
}, 0.001);
tl.staggerTo(bP, 3, {
    cycle:{
        y:[700, -700, -1000, 1000],
        x:[200, -200, -700, 700],
        rotation: function(i) {
            return i * 20
        }
    },
    opacity: 0,
    fill: "#f2bf30",
    ease:Circ.easeInOut
}, 0.001, "start+=1.25");
```

HSL 颜色渐变动画

这个动画是“相对”简单的。HSL（H，色相；S，饱和度；L，明度）颜色的渐变能力是非常奇妙的。当你想创建一些复杂的、呈现为动画形式的颜色效果时，稍微调整这些值就会带来非常强大的视觉效果。

例如，一个场景中存在不同颜色的元素，可以在某个过程中缓慢改变场景中所有元素的颜色，比如从白天到晚上。以前实现这种效果最简单的方法是逐步改变这些元素自身的颜色值。也可以使用一个遮罩层来覆盖整个场景，但这样缺乏真实性。或者可以使用一个 SVG 矩阵滤镜，但这样的效果非常一般，并且很难呈现为动画。还可以使用目前支持度并不高的 CSS 滤镜。

然而，使用 GSAP，一小段代码就可以完成这个效果，并且完美向后兼容。通过 GSAP 可以操作数以百计的元素，让它们稍稍变暗，并且减少它们颜色的饱和度，缓慢调整它们的色调，就像将它们置为阴暗处一样。由于 HSL 颜色渐变不需要依赖一个具体的属性，因此能够用于 background（div）或 fill（SVG）。

图 11-3 和相应的示例（*http://codepen.io/sdras/pen/zvwGKw*）展示了它是如何工作的。

图11-3：当鼠标悬停在乌龟上时，乌龟SVG中的每个形状的图形分别会做色相、饱和度或亮度值的插值改变

这么多选择！有什么好例子吗？可以将交错周期和 HSL 颜色补间放在一起。相比之前的夜景效果，后一个效果更加极端。 126

以下两个不同的按钮，相对效果略有不同。因为是对相对值做补间，所以可以将按钮的效果组合起来并获得多个输出。

```
function hued() {

    var ch1 = "hsl(+=110, +=0%, +=0%)",
    tl = new TimelineMax({
        paused: true
    });

    tl.add("hu");

    tl.to(mult, 1.25, {
        fill: ch1
      }, "hu");

    tl.to(body, 1.25, {
        backgroundColor: ch1
      }, "hu");

    tl.from(gauge, 2, {
        rotation: "-=70",
        transformOrigin: "50% 50%",
        ease: Bounce.easeOut
      }, "hu");
```

```
127     return tl
}

var hue = hued();
```

之后我们将使用 `circle` 属性，添加几个节点使场景效果变得更加细腻。目标是所有进场元素的最终效果看起来一样，所以使用 `.staggerFrom` 比 `.staggerTo` 更适合：

```
tl.staggerFrom(city, 0.75, {
    y: -50,
    scale: 0,
    cycle:{
        x:[300, 100, 200],
        opacity:[0.5, 0.3, 0.2, 0.8],
        rotation:[50, 100, 150],
    },
    transformOrigin: "50% 50%",
    ease: Back.easeOut
  }, 0.02, "in");
```

这样就创建了一系列城市建造的效果，如图 11-4 所示。

在这里，我们将 HSL 补间动画与交互联系起来，通过拖曳来控制时间轴。这样可以创建很多有趣的效果，GreenSock 与用户的操作可以很好地结合起来。我们将在第 12 章讨论这一点。

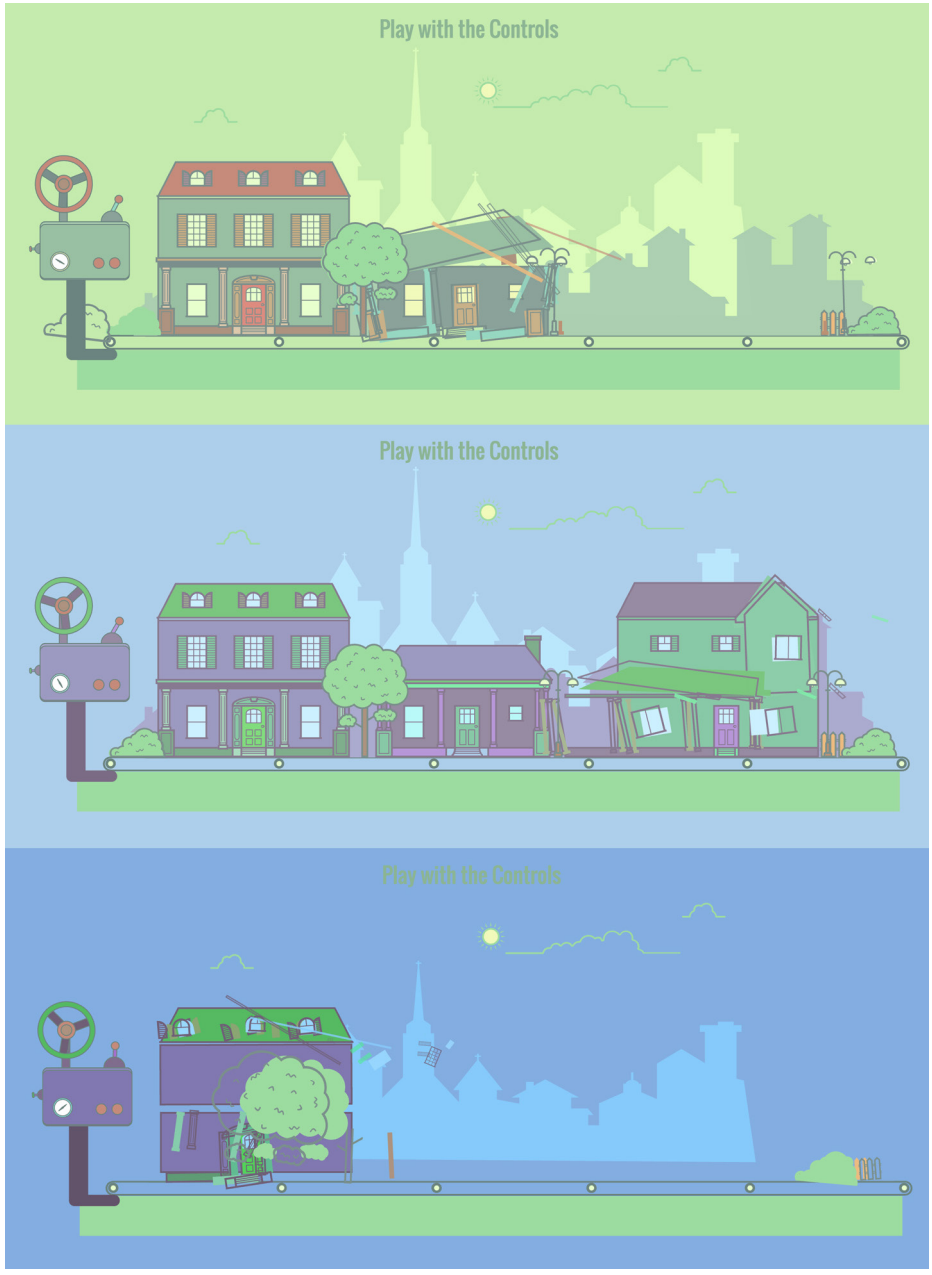


图11-4：可以使用控件来控制所有的填充和背景色，并进行独特的颜色组合

129

文字切分

虽然 SplitText（*https://greensock.com/SplitText*）并不用于 SVG 动画，不过这个效果很适合同 SVG 动画一起提出。不过，本书中介绍这个效果并不意味着这个效果是通过 SVG 的 <text> 节点实现的。

SplitText 兼容 IE 8，独立于 GreenSock。它可以根据选择的内容将文本划分为字符、单词或行，并将其包装到单独的 div 中，以便可以同时或依次操作它们。

下面是一个简单的例子：

```
new SplitText("#myTextID")
```

130 图 11-5 和相关的 demo（*http://codepen.io/sdras/full/RNWaMX*）把 SplitText 作为一个对象来使用，并将其用于获取字母和单词，从而在其上制作动画。

131 代码如下：

```
function sceneOne() {
    var tl = new TimelineLite(),
        mySplitText = new SplitText($text, {
            type: "chars, words"
        });

    tl.staggerFrom(mySplitText.chars, 0.8, {
        opacity: 0,
        scaleX: 0,
        ease: Power4.easeOut
    }, 0.05, "+=4")
    .staggerTo(mySplitText.words, 0.8, {
        rotationY: 60,
        y: 300,
        opacity: 0,
        ease: Power4.easeIn
    }, 0.1, "+=0.1")
    .to(person, 3, {
        rotation: -5,
        transformOrigin: "80% 50%",
        y: -10,
        ease: Circ.easeOut
    })
    .to(head, 3, {
        rotation: -10,
        transformOrigin: "0% 100%",
```

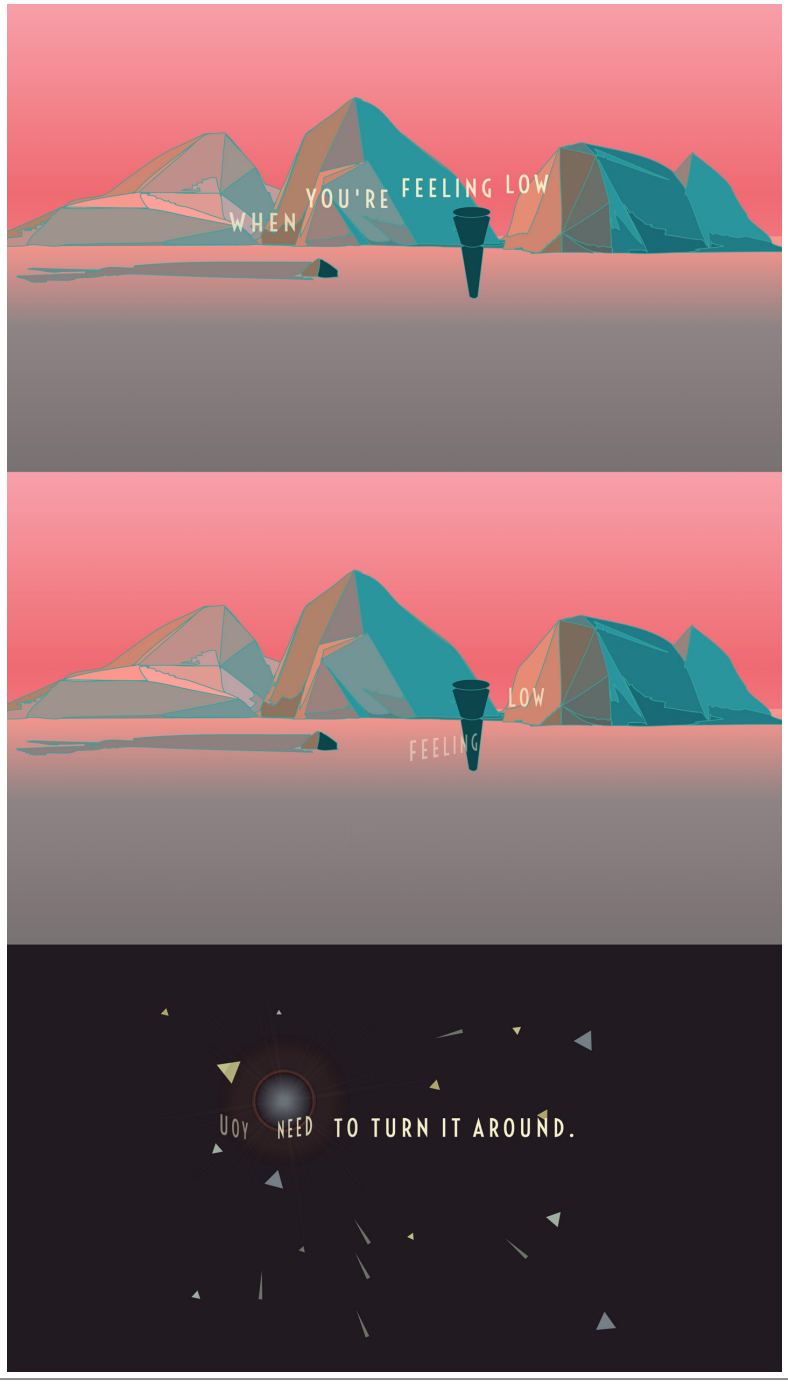


图11-5：动画可以为你的故事增光添彩

```
        y: 10,
        ease: Back.easeOut
    }, "-=3")
    .to(neck, 3, {
        rotation: -10,
        transformOrigin: "0% 100%",
        y: 10,
        ease: Back.easeOut
    }, "-=3");

  return tl;
}
```

SplitText 不是唯一可以创建文本效果的库，但是在这个库中包含一个杀手功能，其他库通常没有，它能够很自然地处理文字换行。

拆分元素也可以将它们的位置设置为“相对”或“绝对”。当使用 position:relative 分割时，文本能够随着父元素更改大小而自然地分散和聚集。当使用 position:absolute时，文本被拆开后不会聚集起来，这可能会提高动画性能。

132 如果需要制作文本动画，且将其返回到“非分割”状态，可以使用 revert() 方法。还可以给每个分解的文本添加自动递增的类名，例如 .char1、.char2、.char3 等。

```
new SplitText("#myTextID", {type:"words", wordsClass:"char++"});
```

这使我们能够创造有趣的效果，甚至针对某个特定单词、字符或行制作动画。

第 12 章 133

DrawSVG和Draggable

Draggable

在 Web 页面中，拖动屏幕上的元素看起来是一件非常简单的事情，就像我们用工具乱涂乱画。要做这件事，可以用：触摸输入设备、鼠标事件、视窗、滚动行为、摩擦和常见的物理现象。当然，也会有很多你开始没有考虑到的会失败的情况。但值得高兴的是，GreenSock 的 Draggable（*https://greensock.com/draggable*）是一个真正有用的插件，它对 HTML 元素非常有用，当然同样也包含 SVG。

Draggable 支持触屏设备，使用 `requestAnimationFrame()` 方法，也支持 GPU 加速。Draggable 自己能独立工作，但是如果和 ThrowPropsPlugin（*https://greensock.com/throwpropsplugin*）一起用会更高效，可以创造真正漂亮的物理动效。

Draggable 最好的一点就是它十分简单。让一个区域元素变得可拖曳所需要做的一共就是如下一行代码：

```
Draggable.create(".box", {type:"x,y", edgeResistance:0.65,
    bounds:"#container", throwProps:true});
```

你可能会发现，在代码的开始部分，我们用 `bounds` 定义了一些边界。`bounds` 相当灵活，你可以定义包含单位或者像素的参数。像 `#container`（在之前的示例中）或者其他块都会有效，但是你会说一个对象的 `{top:10, left:10, width:800, height:600}` 限制了运动。

你也可以限制它只能沿着横轴或者纵轴移动，只要设置 `lockAxis` 为 `true`，这样它就只会在两个方向上运动。

有很多你可以使用的回调函数和事件监听函数，如下所示：

134

- onPress
- onDragStart
- onDrag
- onDragEnd
- onRelease
- onLockAxis
- onClick

这样，你可以像下面这样注册一个 dragend 事件的处理函数：

```
myBox.addEventListener("dragend", functionName);
```

它指向 Draggable 实例本身，你可以简单地获取它的 target 或者 bounds。这非常有用，因为你即将开始使用 Draggable 提供的功能，不需要猜测或者打印日志来理解你正在指向的。所有这些也同样可在变化的元素上生效，比 transform-origin 更优——比起编写原生的方法，插件变得更简单以及更直接。图 12-1 展示了一些与插件相关的示例（*http://codepen.io/sdras/pen/gbERKQ*）。

图12-1：图中是一个真正简单的马铃薯先生，允许你拖动周围的SVG元素来填充和匹配它的容貌

我们真正唯一需要的就是下面这段代码，可让它完全生效：

```
var features = "#top_hat, #moustache, #redhat, #curly-moustache,
    #eyes1, #lips, #toothy-lips, #toupe, #toothy, #big-ear-r,
    #big-ear-l, #shoes1, #lashed, #lash2, #lazy-eyes, #longbrown-moustache,
    #purplehat, #sm-ear-r, #sm-ear-l, #earring-r, #earring-l, #highheels,
    #greenhat, #shoes2, #blonde, #blond-mustache, #elf-r, #elf-l";
```

```
Draggable.create(features, {
    edgeResistance:0.65,
    type:"x,y",
    throwProps:true,
    autoScroll:true
});
```

135

drag 类型

到目前为止，我们只是展示了如何用 Draggable 设置 *x* 和 *y* 的值，代表闪烁和上下、左右移动屏幕里面的东西。还有其他的 drag 类型可以选择：`rotation` 和 `scroll`。

```
Draggable.create("#wheel", {type: "rotation"});
```

在上面的代码中，Draggable 使用了 `rotation` 类型，它很有趣，在控制诸如把手、齿轮、杠杆还有滑轮的时候更炫酷。你还可以定义 `minRotation` 和 `maxRotation`。

我并不倾向为了滚动而使元素混乱，除非必须要使用，但是还是需要提一下，Draggable 对它同样有用。你可以控制一个元素的 `scrollTop` 和 `scrollLeft` 属性；给 `lockAxis` 设置一个布尔值；用数值类型来定义 `edgeResistance`。

我所做的事情是和 `scrolling` 有关的，比如 `autoScroll:1`（或者 `autoScroll:2` 等）。如果你要移动的元素在区域之外，那么这将使视窗出现滚动条。假设在屏幕上拖动一个区域，如果你传入的是 `autoScroll:1`，滚动就会跟随区域的边缘，这也正是你所期待的效果，并且这种效果看起来非常自然。

hitTest()

Draggable 最酷的一个功能是它能进行自定义碰撞检测，这使得拖曳类的 UI 界面和交互变得可能，甚至是类似游戏的东西。

在下面的代码中，我们会检查元素重叠是否超过 80%。如果超过，增加一个类（它会增加一个红色的边框），如果没有，则删除它。

```
var droppables = document.querySelectorAll('.box'),
    overlapThreshold = "80%";

Draggable.create(droppables, {
    bounds:window,
    onDrag: function(e) {
        var i = droppables.length;
```

136

```
            while (--i > -1) {
                if (this.hitTest(droppables[i], overlapThreshold)) {
                    droppables[i].classList.add("red-border");
                } else {
                    droppables[i].classList.remove("red-border");
                }
            }
        }
    });
```

可以使用 `hitTest()` 方法来检测是否发生了一个鼠标移入事件，或者其他类型的事件。

还可以使用一些原生的方法来达到同样的目的，比如 `getBBox()` 方法或者 `getBoundingClientRect()` 方法来计算相邻的值。我会在第 15 章给大家展示如何用这些来开发炫酷的效果。如果你已经加载和开始使用 Draggable 插件来处理拖动事件，那么完全可以使用这些有用的方法集合。

用 Draggable 来控制时间轴

我喜欢的一种方式就是把 Draggable 和其他效果一起用。用 Draggable 绘制一个时间轴可以创造出漂亮的场景，给用户提供掌控过程的能力。来看一下图 12-2 和它相关的示例
137 (*http://codepen.io/sdras/full/NqYGZv/*)。

138 简短地说明一下，在这里我没有展示页面所有元素动画的代码，更多的是专注在如何创造这种交互上：

```
var master = new TimelineMax({paused:true});
master.add(sceneOne(), "scene1");

// master.seek("scene1");

Draggable.create(gear, {
    type: "rotation",
    bounds: {
        minRotation: 0,
        maxRotation: 360
    },
    onDrag: function() {
        master.progress((this.rotation)/360 );
    }
});
```

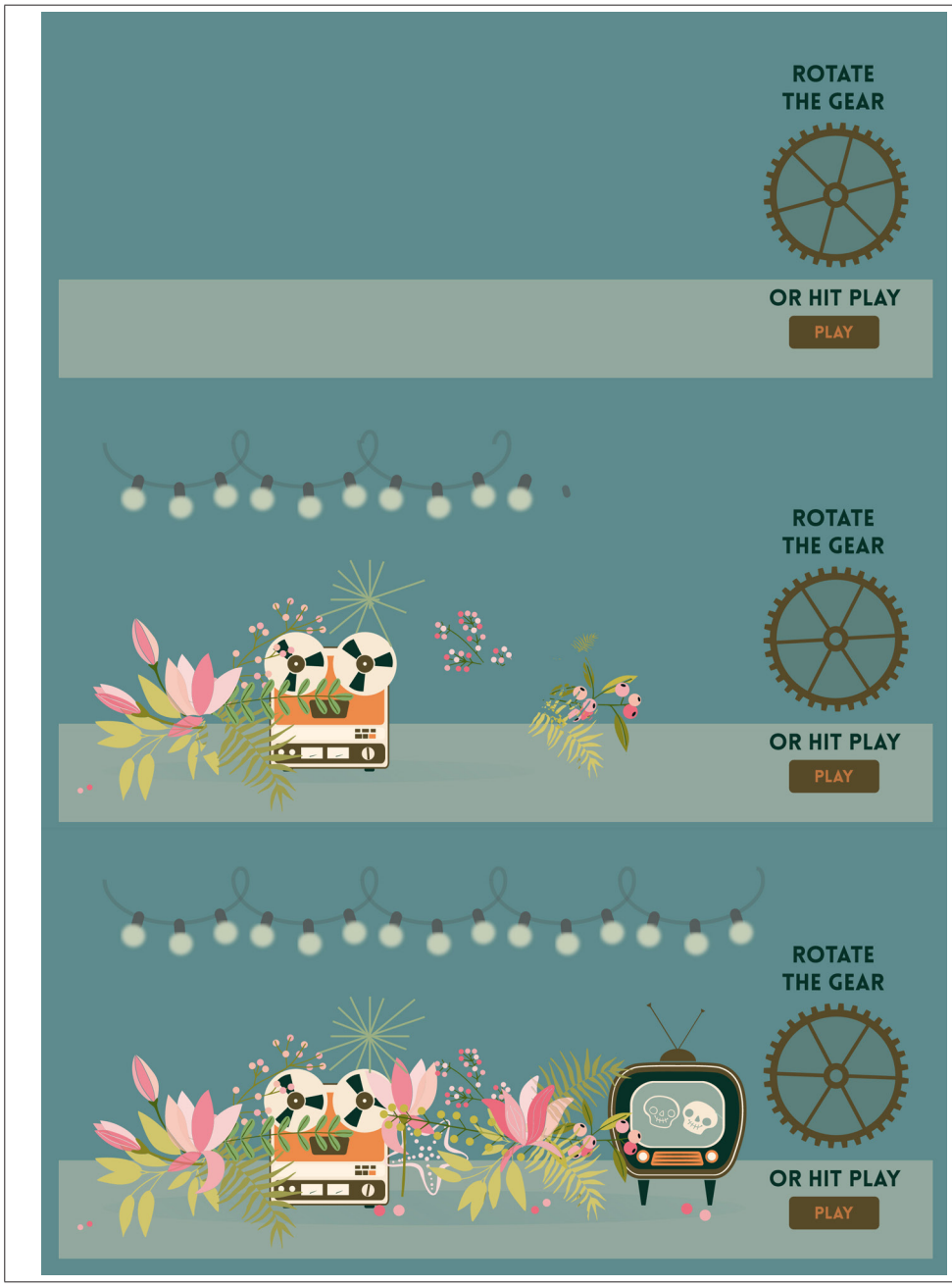


图12-2：用户可以选择不停地转动齿轮或者按PLAY按钮

开始的时候，我们暂停了时间轴，创建了一个齿轮的 draggable 实例。我们用 `minRotation`

和 maxRotation 来定义 bounds，这样用户不能一直拖动齿轮。还指定了时间轴用 progress() 方法沿着旋转方向 360° 地进行绘制。progress() 方法对于这类时间轴操作非常有用，它允许你创建时间轴的范围或者操作沿着时间轴上的某一个点的事件。

你也可以在直接划分交互或者其他任意你想做的事——有很多可能性!

DrawSVG

如何能让一个 SVG 元素看起来像在页面上绘制自己呢？事实上，我们已经学习了这些知识——在第 6 章当我们学习 CSS 动画示例的时候。

加载插件程序

DrawSVG 是一个付费的插件程序，但是在付费之前，可以试用它。GreenSock 让这变为可能，它提供了 CodePen 的插件程序版本（*http://bit.ly/2lv1xhg*），这样你可以在编写示例的时候使用它。

在开始编写示例之前，不要忘记加载插件程序资源和 *TweenMax.min.js* 文件。

我们来回顾一下这项技术。

首先，需要一个 SVG 元素。这个元素有一个虚线边框，如图 12-3 所示。

139 我们可以用 JavaScript 原生的方法 .getTotalLength() 来获取形状的长度。

```
var starID = document.getElementById('star');
console.log( starID.getTotalLength() );
```

我们将设置路径上带有虚线的部分，用 stroke-dasharray 来设置形状的长度（从控制台打印出来的整数）。 offset 或者 stroke-dashoffset 属性放置了路径开始部分和图案之间的间隔。即使只设置了数组中的一个值，它也会自动复制这样的间隔和虚线数量。下面这段动画示例代码，不用任何框架，只用 CSS 或者 JavaScript 代码。

140
```
.path {
    stroke-dasharray: 1000;
    stroke-dashoffset: 1000;
    animation: dash 5s linear forwards;
}
@keyframes dash {
    to {
        stroke-dashoffset: 0;
    }
}
```

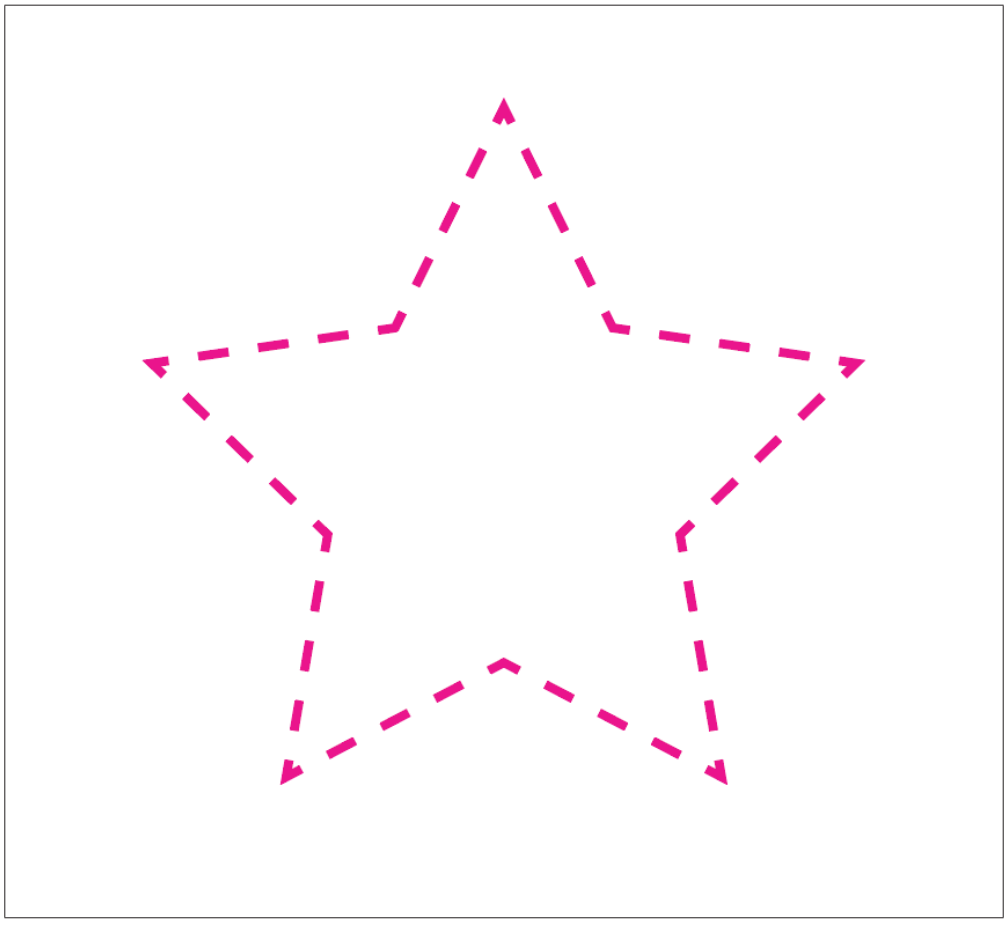


图12-3：以带有虚线 stroke 的五角星形状

所以，为什么我们用 GreenSock 来做动画，而不是简单地用不带框架的 CSS 做动画。这里有一些原因：

- 你已经加载了 GreenSock，插件允许你不计算长度，而轻松地使用这些属性进行动画操作。
- 没有 `getTotalLength()` 方法，也可以用于 `rect`、`circle`、`ellipse`、`polyline` 和 `polygon`。
- `.getTotalLength()` 方法是静态的，当 SVG 响应式地调整收缩的时候，它不起作用，但是这个时候 DrawSVG 可以。
- `.getTotalLength()` 在 IE 和 Firefox 中有一些奇怪的 bug。如果你只是用它们来打印值，然后再删除它，那么没有问题，但是，当用这个方法来动态更新的时候

它不会生效。

- 使用 GreenSock，你不但可以从一开始到结束使用一个整数，还可以使用布尔值（`true`表示完全绘制，`false`表示完全不绘制）或者是百分比。甚至可以以这样的值来处理动画的进入和消失，比如“`50% 50%`”。

DrawSVG 让我们做复杂而又稳定的动画变得十分简单，当它和其他特性一起配合使用的时候，可以简单地创造漂亮的效果，且只需要下面一行代码：

```
TweenMax.staggerFromTo($draw, 4,{ drawSVG:'0' }, { drawSVG: true }, 0.1);
```

图 12-4 展示了一个更复杂的示例（*http://bit.ly/2n7sUSP*），暗示了更多的可能性。

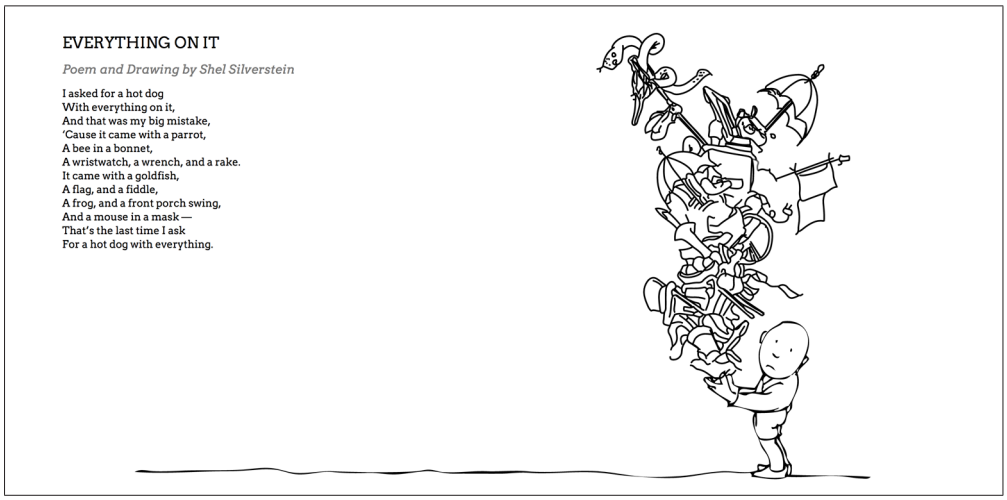


图12-4：示例展示了当文字按照动画效果进入页面的时候，在页面中开始绘制Shel Silverstein

141 DrawSVG 使用起来很简单，它提供了多种方式来处理复杂的需求，也在更大的场景中变得有用。

第 13 章 143

mo.js

mo.js（*http://mojs.io/*）是一个专注动效的 JavaScript 库（截至目前，GitHub 中已经累计有 11 529 个 star 了），它提供以声明式语法来完成元素的动画和动效的功能。虽然 mo.js 还处于测试版本，但是已经展现了很多令人惊奇的功能。这个库的作者是来自微软的 Oleg Solomka（网名是 LegoMushroom），你可以在网上找到很多他通过 mo.js 制作出来的令人惊叹的 demo，以及有关 mo.js 的教程。在本章，我们将对 mo.js 功能做一个快速介绍，帮助你快速上手 mo.js。

mo.js 基础介绍

mo.js 主要提供了两种方式来操纵 DOM 节点运动：

- 通过新建一个对象，在 mojs.html 找到对应的节点，然后让其运动。这种方式和其他动画库的操作方式差不多。
- 通过创建一个特殊的 mo.js 对象（例如，Shape、ShapeSwirl、Burst 等对象），控制声明 DOM 节点。这种对象可以提供一些 mo.js 的特殊功能。

而无论通过哪种方式操控 DOM 节点运动，mo.js 都会提供一些通用且基础的功能。例如，自定义路径的缓动函数和时间轴进度控制等，这是功能强大的动画制作工具，可以让你在工作中更快地适应 mo.js。

图形

mo.js 可以简化使用者的工作，因为 mo.js 提供的声明式语法能够让使用者随心、简单地创造一些图形并使它们运动起来。

图 13-1 所示的是一个学习相关基础语法的 demo 实例（*http://bit.ly/2hiKsG4*）。

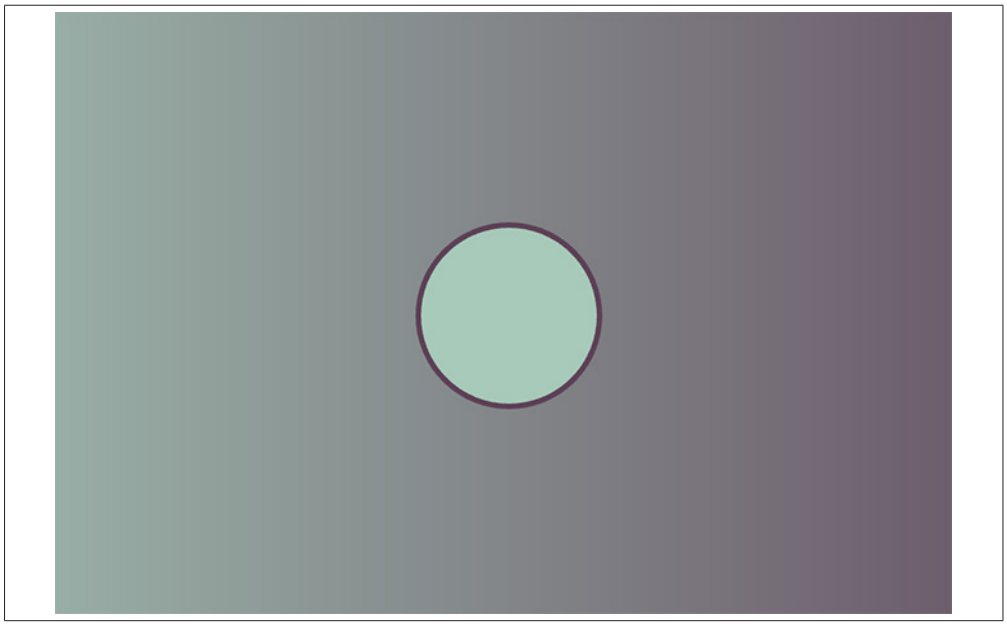


图13-1：一个简单但富含很多细节的圆，作者还对其细心地进行了绝对居中

144 如下是这个例子所对应的代码：

```
var shape = new mojs.Shape({
    shape:          'circle',       // 默认的图形就是圆形
    radius:         50,
    fill:           '#A8CABA',      // 默认颜色是粉色
    stroke:         '#5D4157',
    strokeWidth:    3,
    isShowStart:    true,           // 任何动画运行前都展现的是这个图形
});
```

这里有一些你需要了解的基本知识：

- 可以创建的图形包括圆（`circle`）、矩形（`rect`）、加号（`cross`）、等号（`equal`）、折线（`zigzag`）以及多边形（`polygon`）（默认是圆形）。
- 可以定义图形的填充色、描边色和描边宽度（默认的结果为只填充粉色而不描边。等号和加号没有空间填充颜色，所以必须为它们设置描边色和描边宽度，否则即使创建了它们也是无法显现的）。
- 当图形宽高相同时，可以通过设置 `radius` 来确定图形的宽高。如果需要的图形宽高不同，那么可以分别使用 `radiusX` 来调整宽度，`radiusY` 来调整高度（`radius`

的默认值为 50）。

- 如果你只想展现一个图形，并不准备对该图形做动画，那就必须设置 `isShowStart: true`，使图形能够可见（默认值为 `false`）。
- 对于 `polygon`、`zigzag` 和 `equal`，可以设置它们的 `Points` 属性，即图形点的个数，根据设置不同的 Points，可以创造出多种不同的图形。 145
- 除非你对图形设置特殊的 `top`、`left` 等值，否则所有图形都默认为绝对定位，并且相对于屏幕居中。

图 13-2 所示为上文所对应的图形，对应的 demo 链接地址为 *http://bit.ly/2hfqThp*。

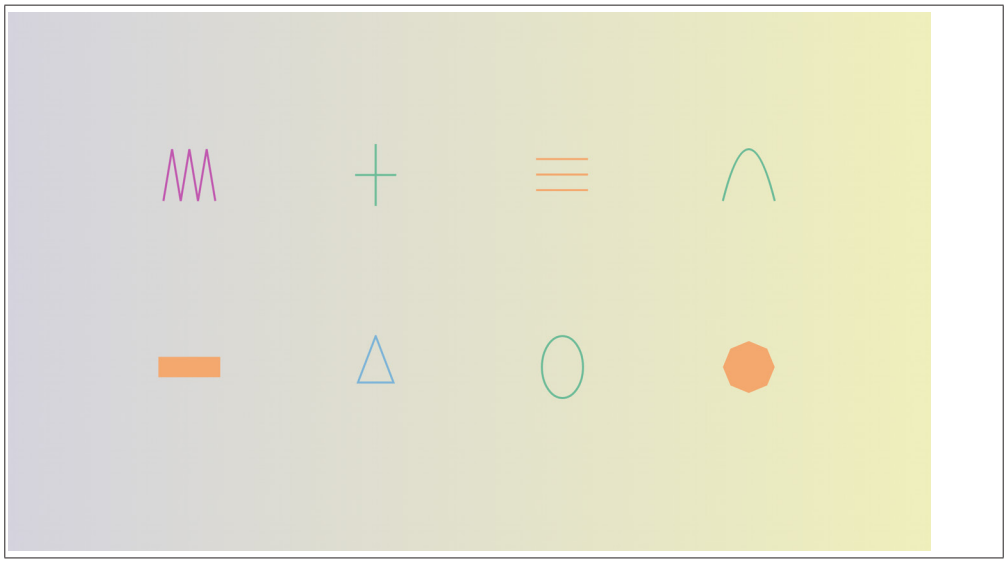


图 13-2: 一些可用 mo.js 创造的图形

下面的代码是图 13-2 左上角折线图形所对应的代码：

```
const zigzag = new mojs.Shape({
    shape:          'zigzag',
    points:         7,
    radius:         25,
    radiusY:        50,
    top:            pos.row1,
    left:           pos.col1,
    fill:           'none',
    stroke:         color1,
    isShowStart:    true,
});
```

如果你通过 Chrome 浏览器工具看了产生图形的 DOM 结构，会发现 mo.js 的原理实际上是通过将 SVG 元素嵌套在一个 div 中，通过控制这个 div 来对 SVG 进行定位，例如控制图形的 top 和 left。可以传递一个已存在的父元素来充当这个 div。例如通过设置 parent 属性：parent: '#id-to-be-placedunder'，当然也可直接传递一个任意的 DOM 对象，例如，parent: someElement。有些时候还可以选择使用 div 或者 SVG，如果能把图形放置在 viewBox 内，那么在创建移动端响应式的场景中创作伸缩动画将会变得更容易。

也可以按照下文提供的方式，自定义一个图形并添加到 shape 对象中来做动画：

```
// 自定义图形
class OneNote extends mojs.CustomShape {
    getShape () { return '<path d="M18.709
    ...
    "/>';
    }
}
mojs.addShape( 'oneNote', OneNote );

const note1 = new mojs.ShapeSwirl({
    shape: 'oneNote',
    ...
});
```

图形的运动

不但可以使用 mo.js 创造图形，而且可以赋予图形动画。怎么赋予图形动画呢？ mo.js 给我们提供了简单的方式：传递一个对象，通过对象的 key、value 来决定图形的运动，如，{fromvalue: tovalue}。

可以类比上文所提到的 from 和 to 的用法。值得注意的是，在 ES5 中，无法直接对一个对象的 key 设置变量，但可以使用变量存储 *fromvalue* 的值。

方案如下：

```
var fromvalue = '123';
options = {scale: {} };
options.scale[fromvalue]='tovalue';

console.log(option);
>>> 输出
```

```
{
    scale: '123': 'tovalue'
}
```

而在 ES6 中，这更容易，因为当声明对象时，可以使用变量来实现 key 值。

方案如下：

```
var fromvalue = '123';
var option = {
    scale: {
        [fromvalue]: 'tovalue'
    }
}

console.log(option);
>>> 输出

{
    scale: '123': 'tovalue'
}
```

还可以声明变形属性，例如，scale（形变）、angle（旋转，可类比 CSS 中的 rotate 属性），以及 opacity（透明度）和 fill。对于 fill 属性，还可以插入颜色，如图 13-3 所示。

```
scale: { 0 : 1.5 },
angle: { 0 : 180 },
fill: { '#721e5f' : '#a5efce' },
```

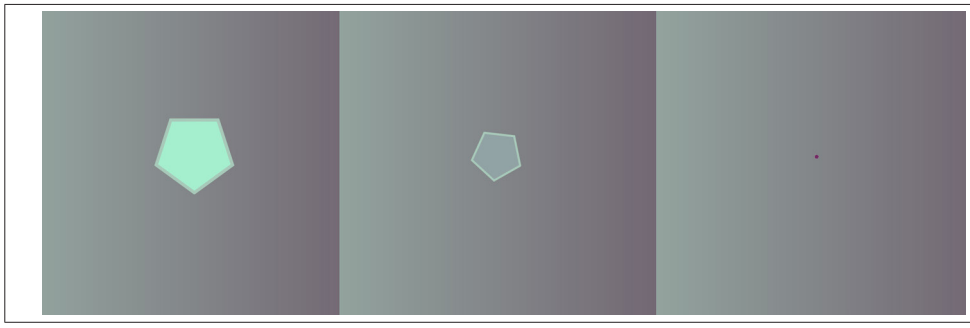


图13-3：通过插入如上代码使图形在两个状态间进行补间动画

除此之外，还可以定义以下几个参数。

147

- 动画时长（duration）
- 动画延时（delay）

- 是否重复播放（repeat）
- 动画播放速度（speed）：默认是 1，设置为 0.5，动画速度减半，设置为 1.5，动画速度变为原来的 1.5 倍。
- 是否来回播放（isYoyo）：布尔值，是否让动画像 yoyo 球一样来回运动。默认为 false。
- easing 设置：在某个属性对象中设置，设置单个属性运动的节奏。
- 返回播放时候的 easing（backwardEasing）：在设置了 isyoyo:true 的前提下，可以设置这个属性，这个属性作用于返回播放时候的 easing（从 1 回到 0）。因为有时候，返回播放的节奏和正常播放的节奏不同，所以可以通过这个值单独设置（如果没有设置，则默认值同 easing 设置）。
- 是否淡出（isSoftHide）：布尔值，默认为 true，即每次隐藏图形的时候都通过改变 scale: 0 进行淡出，如果为 false，则直接 display: none。

使用 mo.js 提供的随机数生成器

我们用其他 JS 库的时候可能需要写一个生成随机数值的辅助函数，或者用一点代码来实现随机数的向上或向下取整（具体可见第 11 章的 Math.random() 注释部分）。

而 mo.js 则将随机数的生成完美地抽象出来了，我们可以通过传递一个字符串来产生一个随机数：'rand(min, max);'。举个例子，angle: 'rand(0, 360)'，将产生一个 0~360 范围内的随机角度。

链式调用

如果做动画的时候你喜欢采用链式调用的方式，那么可以在初始的定义动画中使用 Promise/A 规范的 .then() 语法来链式地声明动画，例如下面这个 demo（*http://bit.ly/2gwtmpo*）：

```
const polygon = new mojs.Shape({
    shape:      'polygon',
    points:     5,
    stroke:     '#A8CABA',
    scale:      { 0 : 1.5 },
    angle:      { 0 : 180 },
    fill:       { '#721e5f' : '#a5efce' },
    radius:     25,
    duration:   1200,
    easing:     'sin.out'
}).then ({
    stroke:     '#000',
    angle:      [-360],
```

```
    scale:      0,
    easing:     'sin.in'
});
```

148

旋涡动画

mo.js 提供了诸如 ShapeSwirl 和 Burst 这样有趣的功能，它们能够在 SVG 上绘出优雅的动画。一个 ShapeSwirl 类似一个常规的图形对象，但是，ShapeSwirl 所产生的图形的运动节奏就和它所叫的名字一样——图形是以旋转的旋涡轨迹运行的。对于旋涡动画（swirl），mo.js 提供了多个参数供我们配置，它们都是用于配置 Swirls 所依赖的正弦函数的参数。

- 旋转的振幅（swirlSize）：在水平方向上，图形旋转运动的轨迹振幅（即设置正弦函数的偏移量或振幅）。
- 旋转的频率（swirlFrequency）：旋转运动的快慢程度（即正弦函数的频率）。
- 路径缩放（pathScale）：正弦函数在 *Y* 轴方向上缩放。
- 角度偏移量（degreeShift）：设置旋涡动画的轨迹方向，默认竖直向上为 0，顺时针方向旋转，设置区间为 [0, 360]（和 Burst 动画一起使用效果更佳）。
- 正弦函数方向（direction）：调整旋涡动画运动轨迹方向，由右到左还是由左到右（即正弦曲线波峰，先上后下还是先下后上），参数只能设置为 -1 或 1（默认为 1）。
- 是否使用漩涡动画（isSwirl）：布尔值，默认为 true，设置为 false 后，则取消旋涡动画。

上面的内容确实有点难以理解，所以我做了一个 demo（*http://codepen.io/sdras/full/mrZWqg/*）来帮助你手动调整这些参数，以便更好地明白这些参数的含义。图 13-4 所示的是 demo 动画的截图。

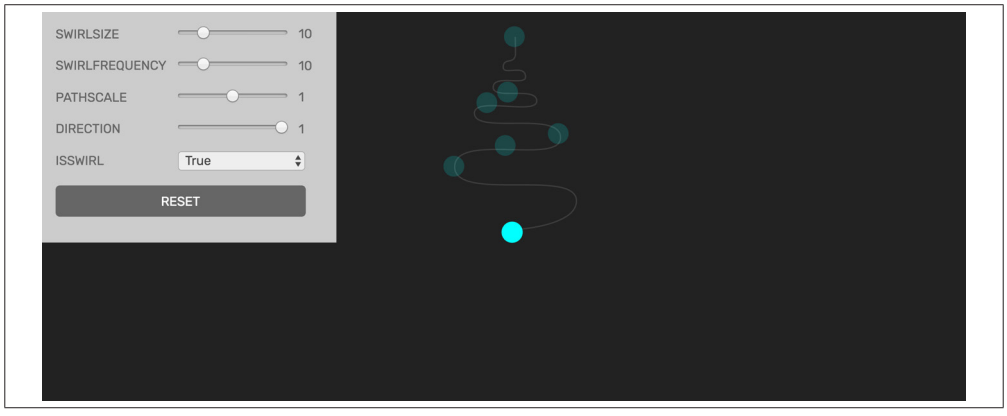


图 13-4：一个可以控制参数的demo，通过这个demo便于理解 mo.js 的ShapeSwirl 的工作原理

149 同时，对于一些常规的配置或自定义的图形对象，我们可以将它们统一抽象出来，和 ShapeSwirls 所需要特殊声明的参数对象进行分离。然后通过 ES6 提供的对象解构运算符来进行合并，这样统一抽象出来的常规配置项就可以进行复用。像图 13-5 所示的示例（*http://codepen.io/sdras/pen/OReWOw*）一样。

```
const note_opts_two = {
    shape: 'twoNote',
    scale: { 5 : 20 },
    y: { 20: -10 },
    duration: 3000,
    easing: 'sin.out'
};

const note1 = new mojs.ShapeSwirl({
    ...note_opts_two,  // 如果不懂 ES6 对象解构语法的读者可以参考
                       // http://es6.ruanyifeng.com/#docs/object 中“对象的扩展运算符”一章
    fill: { 'cyan' : color2 },
    swirlSize: 15,
    swirlFrequency: 20
    }).then({
    opacity: 0,
    duration: 200,
    easing: 'sin.in'
});
```

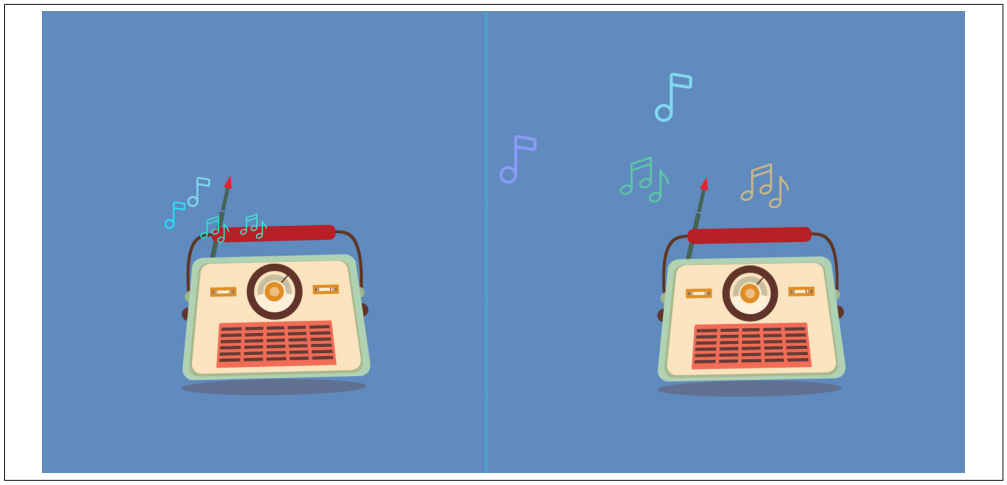


图13-5：上文所提到的使用了ShapeSwirl特性的 demo 效果

爆炸式的效果

一个爆炸式（burst）的效果真的会让人眼前一亮，下面是一个使用默认声明配置爆炸式对象的例子：

```
const burst = new mojs.Burst().play();
```

关于爆炸式的效果，可以对下面介绍的这些选项进行配置。我在这里稍微做了一些配置，写了一个 demo（*http://codepen.io/sdras/pen/kkqNYK*），如图 13-6 所示。 150

- 数量（count）：爆炸出来的子元素的数量。
- 角度（degree）：子元素散开的角度，默认是向 360° 散开。如果设置为 45，那么所有的子元素就以圆心角 45° 区域散开。
- 子元素运动的区间（radius）：语法是 radius: {起始距离 : 终点距离}，如果起始距离为零则从圆心出发。
- 是否渐进隐藏（isSoftHide）：布尔值，决定是否使用 scale: {1: 0} 来隐藏子元素，默认为 true。如果设置为 false，则以 display: none 直接隐藏。这个属性适用于所有种类的图形，但是我再次提及它是因为，这个属性对于爆炸式动画中的某些子元素非常有用。

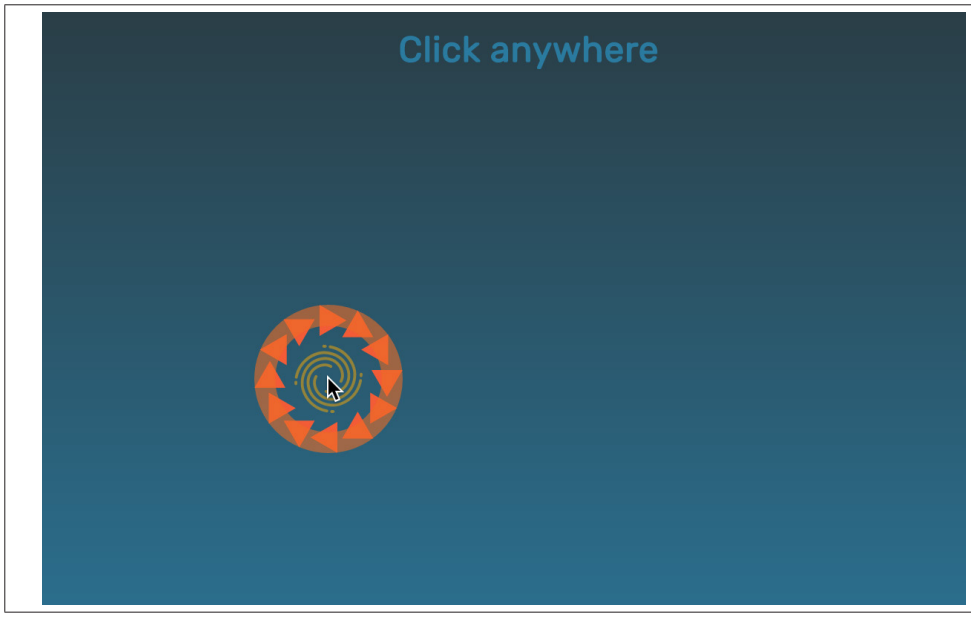


图13-6: 可以在Burst中定制大小、颜色、事件，甚至是自定义形状

所有对于 Shape 图形的参数设置，都适用于 Burst 动画对象。也可以对爆发出来的子元

素单独设置形状，例如下面的例子：

```
const burst = new mojs.Burst({
    radius: { 0: 100 },
    count: 12,
    children: {
    shape: 'polygon',
    ...
    }
});
```

151 时间轴控制工具

通过时间轴工具，可以以显式声明的方式通过 .add 方法来绑定配置对象或者 tween 对象，这样就允许动画按照 .add 的顺序依次执行。你还可以使用 .append 方法把 tween 对象放在运行顺序的末尾。下面是一个有关 Timeline 的例子：

```
const timeline = new mojs.Timeline({
    .add( tween )
    .append( tween )
});
```

.add 允许在时间轴中添加任何动画对象或者 Shape 图形对象。这些动画对象虽然会立即按顺序执行，但是也可以为它们设置 delay 或者 stagger 来调整运行时间。.append 方法和 .add 方法一样，但是顺序上会错开 .add 方法添加的顺序（排在末尾）。

能够对 Timeline 进行配置的参数很少，而且配置这些参数后往往都没有什么价值，和上文介绍的配置方法类似，以对象的形式传递这些参数：repeat、delay 和 speed，下面是一个配置的例子：

```
new mojs.Timeline({
    repeat: 3,
    isYoyo: true
});
```

Timeline 除了可以传递 Tween 对象外，还能够无限制地嵌套多层子 Timeline：

```
const subTimeline = new mojs.Timeline();

const master = new mojs.Timeline()
.add( subTimeline );
```

补间动画

虽然我们在上文中已经介绍了很多 mo.js 提供的动画构造函数，但是还没有详细讨论过关于补间动画的相关知识。tween 即是补间动画，它的主要作用是在一个设定的时间段内，通过改变节点的各种属性来不断更新图形运动的细节，从而产生从一个状态到另外一个状态的过渡。Shape 对象中的很多参数（duration、repeat、easing 等）都能直接在补间动画中使用。这里有一个关于补间动画的简单例子：

```
var thingtoselect = document.querySelector('#thingtoselect');
new mojs.Tween({
    duration: 2000,
    onUpdate: function (progress) { // progress 返回的是 [0,1] 的小数值
        square.style.transform = 'translateY(' + 200*progress + 'px)';
    }
}).play();
```

上面的代码能使一个元素从左到右运动 200px。

在复杂的镭射枪动画 demo（*http://codepen.io/sdras/pen/JRQXGz*）中就是依靠这个原理实现左边两段镭射光线加载的动画的。如图 13-7，这里使用了 <path> 产生的缓动函数来改变动画的运动节奏。关于如何使用 <path> 产生缓动函数，我们会在下一章讨论。同时，关于镭射枪的液体动画，也同样是使用上文所述的原理，不断刷新水波 <path> 的 d 属性来产生水波运动的效果的。

152

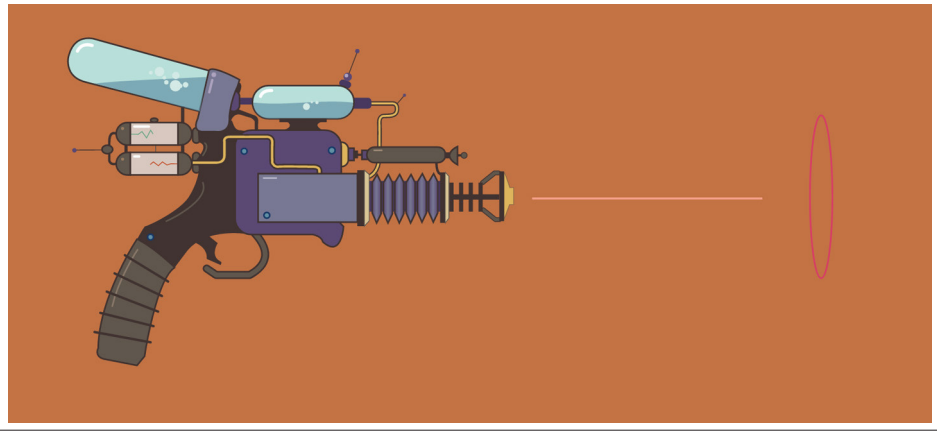


图 13-7：镭射枪动画

下面来看一下上文所提到的两段镭射光线加载的代码：

```
// 镭射枪动画的部分代码
```

```
new mojs.Tween({
    repeat: 999,
    duration: 2000,
    isYoyo: true,
    isShowEnd: false,
    onUpdate: function (progress) {
        var laser1EProgress = laser1E(progress);
        for (var i = 0; i < allSideL.length; i++) {
            allSideL[i].style.strokeDashoffset = 20*laser1EProgress + '%';
            allSideL[i].style.opacity = Math.abs(0.8*laser1EProgress);
        }
    }
}).play();
```

补间动画提供了很多回调函数，例如，监听动画开始和反复开始的回调函数（`onStart` 和 `onRepeatStart`）、完成和反复完成的回调函数（`onComplete` 和 `onRepeatComplete`）、回放开始和回放暂停（`onPlaybackStart` 和 `onPlaybackPause`）等，可以针对动画的不同角度来对动画细节进行微调。如果想了解这些回调函数，可以参考关于补间动画的文档（*http://bit.ly/2lKXq1z*）。

路径函数

这是 mo.js 提供的一个非常酷的功能，它允许输入一段 SVG path 绘制的数学曲线图，然后将曲线图转换为对应的 easing（类比 CSS3 中的 cubic-bezier）。本章中的很多例子都用到了这个功能，但说实话，我永远没法像 LegoMushroom（mo.js 的作者）那样写出那么华丽优雅的教程（*http://mojs.io/tutorials/easing/path-easing/* 教程中用到了可视化图表的形式，很直观地讲述了细节）。我在这里简单地解释一些基础知识并演示一下这个功能是怎么用的，方便大家入门。但是我强烈建议你去看看他所写的教程博文。

153

在我们学习所有的路径 easing 之前，很有必要先夯实一下 easing 最基础的使用方式。可以在构造函数的参数中这样声明一个 easing：

```
easing: 'cubic.in'
```

为什么要学习 easing？关键的原因是，easing 能为你的动画增添活力，它允许通过自定义路径的形式，灵活调整动画的节奏。如果你用 CSS 动画用得很舒服，那么应该也会喜欢上 mo.js 的 `bezier` easing，bezier easing 和 CSS3 中 cubic-bezier 所接受的 bezier 函数是一样的（但 mo.js 却不像 CSS3 cubic-bezier 那样存在某些限制）。这里有一个关于 bezier easing 的例子（*http://cubic-bezier.com/*）：

```
easing: 'bezier( 0.910, 0.000, 0.110, 1.005 )'
```

如果想对 easing 进行更精确的控制，使用 SVG path easing 是一个非常不错的选择。可以传递一个 SVG path 对象，然后 mo.js 会将你传入的 path 转换为 easing，运动图形的节奏就会跟着这个 easing 的输出的百分比进行变化。回顾一下上文截取的镭射枪 demo 中的代码，那里就使用了 path easing 对 onUpdate 输出的百分比值进行了处理：

```
const laser1E = mojs.easing.path('M0,400S58,111.1,80.5,175.1s43,286.4,
                63,110.4,46.3-214.8,70.8-71.8S264.5,369,285,225.5s16.6-209.7,
                35.1-118.2S349.5,258.5,357,210,400,0,400,0');

new mojs.Tween({
    repeat:   999,
    duration: 2000,
    isYoyo: true, // yoyo 反复动画
    isShowEnd: false,
    onUpdate: function (progress) {
        // 将正常的 [0,1] 输入 easing，从而改变动画的运动节奏
        var laser1EProgress = laser1E(progress);
        for (var i = 0; i < allSideL.length; i++) {
            // 使用产出的百分比值，不断刷新，从而改变 strokeDashoffset，
            // 产生线条从有到无再到有的效果
            allSideL[i].style.strokeDashoffset = 20*laser1EProgress + '%';

            // 同时改变线条的透明度，增强效果
            allSideL[i].style.opacity = Math.abs(0.8*laser1EProgress);
        }
    }
}).play();
```

可以通过下一章将要介绍的 mo.js 所提供的曲线编辑工具来切身感受一下 path easing 在运动行为上产生的效果。你可以查看下一章中提供的一些例子。曲线编辑工具能够可视化且直观展现所创造的 easing，这样能够更加精确地调整所创造的 easing。

mo.js 提供的辅助工具

在 mo.js 提供的功能中，令人印象最深刻的就是它提供的辅助工具。为了引起大家的兴趣，这里有一个 Vimeo 网站上使用的示范（*https://vimeo.com/185587462*），如图 13-8 所示。

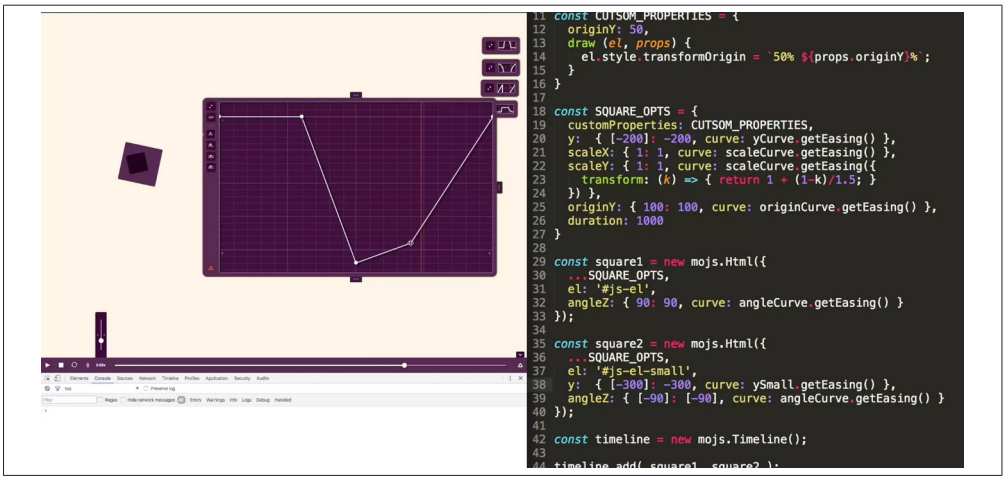


图13-8：Vimeo 上所演示的使用mo.js的工作流程

154 我做了一个可以快速上手的 demo 来展示 mo.js 所提供的动画播放器和曲线编辑器，如图 13-9 所示。图 13-8 左边所示的是曲线编辑工具，藏在下面的、有一个播放按钮的则是动画播放器。大家可以选择把 demo fork 下来或者在线尝试（*http://bit.ly/2gqz2mo*）。曲线工具在工具面板的左边，而时间轴在下端被收起了（单击小箭头按钮可将其展开）。

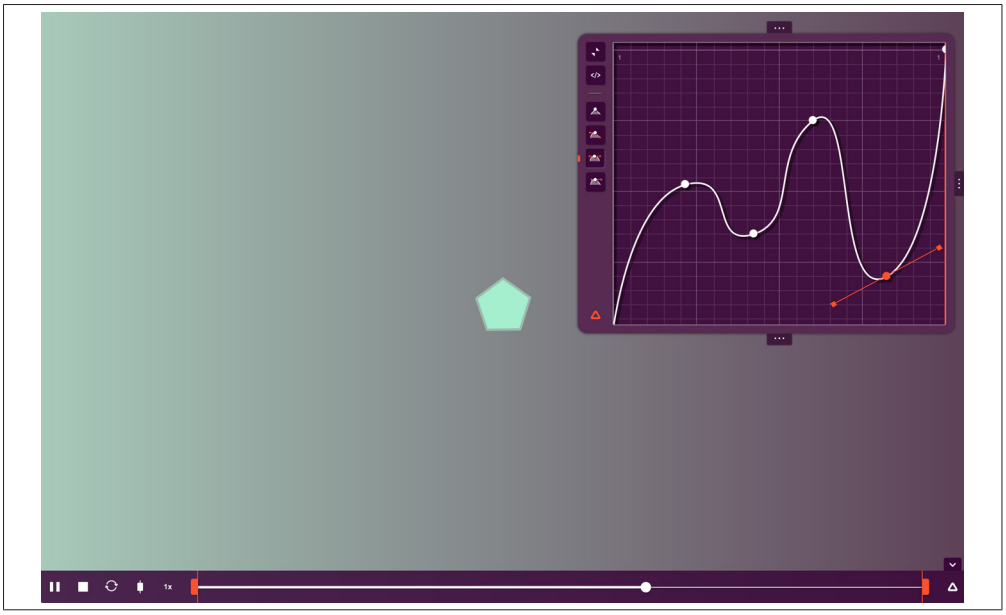


图13-9：这是一个入门的demo，可以使用mo.js的工具简单地调整尝试

现在有一件很酷的事情，LegoMushroom 需要帮手了。他正在对现有的时间轴工具做扩展组件，他已经在 GitHub 上提供了 UE 图（*http://bit.ly/2mtJuZN*）。如果你有兴趣贡献开源项目的话，现在就有一个能让你深挖 mo.js 技术并帮助 LegoMushroom 实现一个有用的辅助工具——只要单击右边的“help wanted”即可！ 155

下面是已完成的、可用的工具的项目地址：

- 动画播放器：*http://bit.ly/2lv7wT9*
- 曲线编辑器：*http://bit.ly/2mHhW2f*

如果你对 mo.js 抱有强烈的兴趣并想贡献代码和深入学习，也可以加入这个社区（*https://hamsterpad.com/chat/mojs*）。

如果你不太喜欢阅读而是喜欢上手操作，这里提供了这章提到的所有 demo 列表（*http://codepen.io/collection/XOEKow/*）。

关于 mo.js 的知识，本章不可能全部涵盖，只是介绍了一些我认为 mo.js 提供的最有用的功能。但是如果想更深入地学习 mo.js，可以参考 mo.js 的官网和 mo.js 的文档（*http://bit.ly/2lkK6ie*）。

157

第 14 章

React-Motion

在 React 中，有很多做 SVG 动画的方法，我们已经介绍过的任意一种技术都需要更改才能用在 React 环境中。React-Motion（*http://bit.ly/2lSUy2l*）有很多卓越的特性，值得我们花一章的篇幅来研究这些特性。

我们在第 7 章提到过，React-Motion 和基于序列的技术，比如 CSS 或者 GreenSock 的时间轴有所不同。在序列技术中，我们并没有真正使用时间来控制插值。

在基于游戏的物理运动中，我们通过 React-Motion 给元素 mass 和 spring 传递参数，然后让它们运动起来。这样，可以得到一个类似现实的*阻碍运动*（后面会解释这个词），并且可以在 UI 中创建非常美妙的动画。

React-Motion 提供了 3 个主要的组件：`<Motion />`、`<StaggeredMotion />` 和 `<TransitionMotion />`。

完整的组件列表如下：

- `spring`
- `Motion`
- `StaggeredMotion`
- `TransitionMotion`
- `presets`

我们在本章中主要研究 `<Motion />` 和 `<StaggeredMotion />`，不过，在该项目的 README 文件中有更详细的介绍（*https://github.com/chenglou/react-motion*）。

加快速度

158

如果你不熟悉 React 或者 ES6 的话，本章对你来说学起来可能有点吃力。我的建议是，先去了解一下它们的工作原理（这超出了本书的范围），然后再回到这里学习 React-Motion 动画库。你不能脱离 React 去使用 React-Motion，所以，先了解基本内容比往下继续更重要。

<Motion />

我们将以 React-Motion 提供的 `<Motion />` 组件开始讲解。

在这些例子中，我们将使用通过 style 传入的整数去改变某些事物的外观。事实上，你可以改变任何路径中的参数或者颜色值。最常见的是使用 style 模式。当然，浏览器也擅长在不触发太多重绘的情况下改变一些 style 的属性值，就像我们在第 2 章中说过的。所以，我们将利用该特点。

在直接看 demo 代码之前，让我们一步一步地分解具体实施的过程。在该例子中（*http://bit.ly/2iFJCG7*），我们将更新一些整数，你将看到发生哪些变化，如图 14-1 所示。

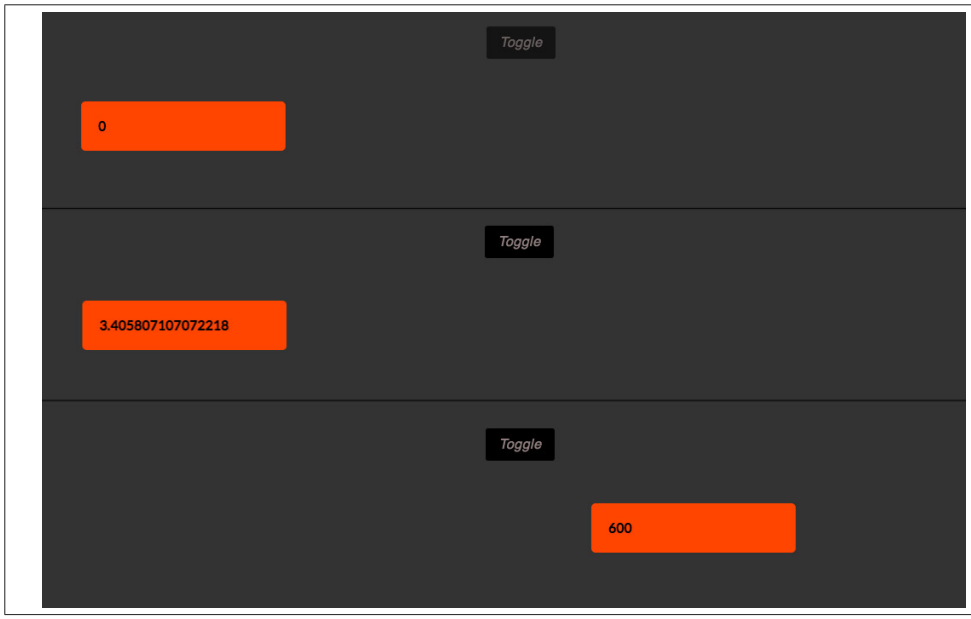


图14-1：插入一个数字，产生一个变化

首先，初始状态是一个简单的布尔值：

159

```
getInitialState() {
```

```
    return {open: false};
},
```

然后，通过鼠标单击或者触碰按钮更改状态：

```
handleMouseDown() {
    this.setState({open: !this.state.open});
},

handleTouchStart(e) {
    e.preventDefault();
    this.handleMouseDown();
},
```

在 render 函数中，我们定义了一个按钮，它会调用刚刚定义的方法去改变状态：

```
<button
    onMouseDown={this.handleMouseDown}
    onTouchStart={this.handleTouchStart}>
    Toggle
</button>
```

然后，我们将通过 React-Motion 使用 `<Motion />` 组件，同时更新 div 上的整数和变形样式。注意，任何时候你使用一个库，比如 GSAP，其实都是通过 JavaScript 来设置内联样式。这并不难，即使它们以另外一种方式书写：

```
<Motion style={{x: spring(this.state.open ? 600 : 0)}}>
    {
        ({x}) =>
            <div className="simple-demo" style={{
                WebkitTransform: `translate3d(${x}px, 0, 0)`,
                transform: `translate3d(${x}px, 0, 0)`,
            }}>{x}</div>
    }
</Motion>
```

我已经将语法代码放在不同的行中，这样你可以更清楚地进行了解。我们将创建一个 `style` 对象，它以 `x` 作为键（可以使用任何内容），并且通过状态的 `open` 属性为 `true` 或者 `false` 的三元操作符来进行赋值。

接着，我们向下传递 `x` 的值，并且能够将它作为一个变量存储任何我们期望的插值。当设置样式时，我们使用 ES6 模板字面量来增加其可读性。注意，这不像 GreenSock，需要手动去处理前缀。你需要手动地将需要的前缀写进去，比如对于 `transform` 属性的 WebKit 前缀。

我也已经将 x 变量放置在 div {x} 中，所以在 div 中通过样式进行移动的过程中，你可以看到该数字的更新。

160

颜色和 React-Motion 插值

与一些基本样式，比如位置属性、SVG 路径数值、透明度等不同的是，某些属性不能接受纯数字作为值。颜色属性就是典型的例子。你需要使用十六进制数或者 RGBA 的值。不过，有很多解决办法。你既可以在数字后面加上 %，也可以通过 `Math.floor()/Math.round()` 来实现。还可以使用 `hsla()`，它代表了一个完整的色相旋转，而且当超过 360° 或者 `s`、`l` 的百分数的范围时，它不会失败。十进制数在将来也会用在颜色组件中，因为 Web 页面的颜色范围已经超出 sRGB 到 DCI-P3 和其他更宽泛的色域了。

现在，基础内容讲完了，我们开始正式做一个 SVG 动画!

看一下图 14-2 及其对应的示例（*http://codepen.io/sdras/pen/ZWeJem*）。

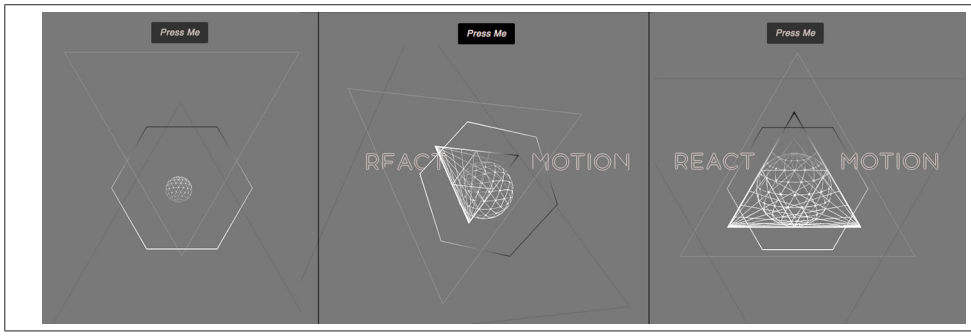


图14-2：当切换的时候，SVG 向里旋转，文字显现

这里有实际运行例子的精简版代码，这样，你可以更清楚地看到它实际的运行原理：

```
<Motion style={{
    // 通过三元操作符，给定所有插值中不同的值
    ...
    dash: spring(this.state.compact ? 0 : 200),
    rotate: spring(this.state.compact ? 0 : 180),
    ...
    }}>
    {/* 保证值能够向下传递 */} {({dash, rotate, ...}) =>
        <svg viewBox="0 0 803.9 738.1" aria-labelledby="title">
            <title>React-Motion</title>
            <g>
                <path
                  style={{
                  WebkitTransform: `scale(${scale}) rotate(${rotate}deg)`,
```

161

```
                transform: `scale(${scale}) rotate(${rotate}deg)` }}
                className="polygon cls-2"
                d="M529.8,359.7l-25.1-43.5-25.4-43.9-25.7-44.4L428,183.3..." />
            </g>
            ...
            <g style={{ strokeDashoffset: `${dash}` }}
                className="react-letters"
                data-name="react motion letters">
                   <path className="cls-5" d="M178.4,247a2.2,2.2,0,1,1-3.5,
                   2.6l-6.5-8.7h-8.6v7.4a2.2,2.2,0,0,1-4.4,0V220.1a2.2,2.2,
                   0,0,1,2.2-2.2h10.8a11.5,11.5,0,0,1,4.8,22Zm-18.6-10.3h8.6a7.3,
                   7.3,0,0,0,0-14.7h-8.6v14.7Z" transform="translate(3.1 1.5)" />
                   ...
            </g>
        </svg>
    }
</Motion>
```

在这里，我们使用完全一样的逻辑，通过一个切换事件来改变状态值（该时间主要参考键 `compact` 或者 `this.state.compact`）。你可以看到是如何在 `<Motion />` 组件中更改不同样式的属性值的，实际是通过传入一系列不同的值。比如，和在第 6 章和第 12 章中介绍的方法一样，对于 `stroke-dashoffset`，我们将使 `dashoffset` 和图形的长度值一样，在该例中，就是 200：

```
dash: spring(this.state.compact ? 0 : 200)
```

`rotate` 将会在 180° 之间交替：

```
rotate: spring(this.state.compact ? 0 : 180)
```

现在我们可以利用 spring 的方式去设计状态样式的改变。我们可以像前面的例子描述的那样，将它用于任意路径，甚至是组。我们通过 JSX 直接将 SVG 内联在代码中。在版本 15 中，所有的 SVG 属性都已经被支持，主要参考 zpao（*https://github.com/facebook/react/pull/6243*）的 PR（Pull Request），我非常感谢他做的工作。不过，唯一的例外是，我们已经将该部分多次重复使用的 `gradient` 直接嵌套在 HTML 中，因为这样可以提高渲染速度和性能（因为我们没有使用 `React.createElement()` 来包裹每一个独立的标签）：

```
<svg width="0" height="0" xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink" viewBox="0 0
                        803.9 738.1">
    <defs>
        <linearGradient id="linear-gradient" x1="399.74" y1="370.41"
          x2="399.74" y2="134.33" gradientUnits="userSpaceOnUse">
          <stop offset="0" stop-color="#fff"/>
```

```
        <stop offset="1"/>
      </linearGradient>
      <linearGradient id="linear-gradient-2" x1="406.42"
        y1="415.63" x2="406.42" y2="166.91" xlink:href="#linear-gradient"/>
    </defs>
</svg>
```

你也注意到了，我将包裹 gradient 的 SVG 容器的高度和宽度设为 0，这样 gradient 就不会对任何 DOM 进行渲染，不过，某些浏览器可能会在该处留下一定的空间。所以为了安全起见，我们得额外实施一些保障措施。通过调用 JSX 中 SVG 元素的 #linear-gradient 的 ID，确保当想调用的时候可以调用。

性能问题

通过 JavaScript 来渲染 SVG 有点耗费性能。我推荐的是，如果 SVG 只有几 KB，那么这看起来问题不大。如果体积很大的话，你应该在 JSX 中复用 <use> 标签。但是，<use> 标签存在一些奇怪的问题，所以，最好使用 <use> 去复用一些静态的图标或者图片。我个人的喜好是通过状态的改变来移动内联的 SVG，但是我也很希望其他人也喜欢这样做。无论你想做什么，要记得实际测试，比如通过 JS 开发者工具中的时间轴面板。

可中断的动画

在上面的例子中，你需要注意到一件事，可以在动画运行的过程中，使其往回运动：这是可中断的。回想一下，我们介绍的其他技术都是依赖时间的。但是，React-Motion 则和时间变量没什么关系。

> 通过持续时间和曲线参数化的动画 API 与连续的流体交互性基本相反。
>
> —Andy Matuschak（*http://bit.ly/2lQv5Jf*）

该库的机制和上面提到的以游戏为基础的物理运动，让我们可以进行中断操作。在一个简单的切换中，这看起来挺简单的。但是如果你准备实现一个菜单的打开和关闭呢？当用户在菜单打开时，决定关了它，并且不能不等待关闭完成，这时又单击了一下菜单，这样可能会出错。这种情况虽然有点少见，不过值得注意。

163

那这意味着可中断的运动更好吗？一方面，是的；另一方面，不是。基于时间和可中断的动画各有各自的用处。

在其他我们已经尝试过的例子中（比如，第 11 章中的文字拆分例子（*http://codepen.io/sdras/pen/RNWaMX*），“扭转它”）或者我的 Pen（*http://codepen.io/sdras/pen/dPqRmP*）上的例子，“当你是个内向的人”，这类流体动画并没有多大意义。在写一个基于时间和序列的动画时，没有使用相关工具去调整具体的序列，这对于你来说有点得不偿失。

另外，如果你有大量的聊天信息，必须在顶部移动，为了能继续输入信息，使用 React-Motion 是最佳的，或者使用其他类似的库。它不仅能让动画运行得很好，而且还能得到一个较好的效果。

`<StaggeredMotion />` 组件显示了这种类型的动效的好处，所以我们将深入了解一下这个组件。

<StaggeredMotion />

像我之前描述的补间动画那样，那效果可以总结为：“这有一组元素。我想通过更新一个属性，让它们在一定时间内发生变化。但是，我们想让它们一个接一个地开始。”React-Motion 中的 `<StaggeredMotion />` 组件并不是完全按照这种方式运行的。

我们已经学习了如何不使用基于时间的序列。那应该怎样让这些事件连续触发呢？特别是当运动是可中断的？具体来说就是我们发送一些东西，然后根据第一个元素的位置，更新下一个兄弟节点的位置。

像这样，当拖曳元素并且更新它的位置时，其他元素并不是直接跟着它，而是以一种十分漂亮的方式陆续完成。如果你再结合弹跳特性（后面会介绍），效果会更好。具体例子可以参考图 14-3 和对应的实际例子（*http://codepen.io/sdras/pen/pyedJE*）。

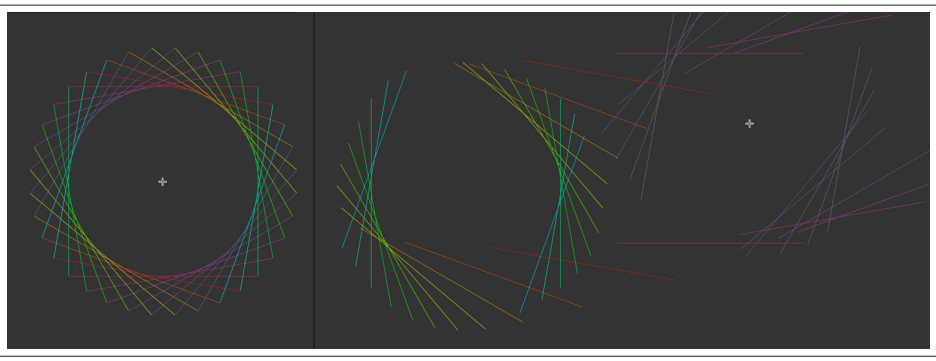


图14-3：当拖曳交错组件时，可以感觉到这是一个可中断的运动，这是和错位方式的差异

如果你只能了解本书中唯一一个例子，那么应该看看这个例子，因为它非常难以描述， 164
除非你亲自看见它。接下来，我们看一下具体的代码。

在 getInitialState() 方法中，我们有一个圆形对象，其 *x* 轴（pageX）的坐标为 250，*y* 轴的坐标为 300。这样做的一部分原因是为了让观看者直接看到级联现象，当他们第一次浏览页面并且圆的位置不在页面的左上角。还可以将 rotate 设为 0。

当加载完组件时，我们开始监听鼠标和触屏输入（对于桌面和手机端）。当事件被触发时，调用不同的函数将圆的 *x* 和 *y* 坐标状态变为触发位置的中心：

```
getInitialState() {
    return {x: 250, y: 300, rotate:0};
},

componentDidMount() {
    window.addEventListener('mousemove', this.handleMouseMove);
    window.addEventListener('touchmove', this.handleTouchMove);
},

// 将 state 和具体的位置绑定在一起
handleMouseMove({pageX: x, pageY: y}) {
    this.setState({x, y});
},

handleTouchMove({touches}) {
    this.handleMouseMove(touches[0]);
},
```

在下一个方法 getStyles() 中，我们根据前一个样式的位置来设置新的样式。注意，这 165
并不是在 render() 方法里做的，需要将原来的样式作为参数传递进去。

我们还将做一些没能在 <Motion /> 组件中做的事（如果你想做，也可以做）：指定硬度和湿度。这可以改变 React-Motion 的弹性运动的效果，以及让其兄弟节点逐个继承该运动。

```
getStyles(prevStyles) {
    // 使用之前的样式改变下一个元素的位置
    const endValue = prevStyles.map((_, i) => {
        let stiff = 200, damp = 15;
        return i === 0
            ? this.state
```

```
            :{
                x: spring(prevStyles[i - 1].x, {stiffness: stiff, damping: damp}),
                y: spring(prevStyles[i - 1].y, {stiffness: stiff, damping: damp}),
                rotate: spring((i * 10), {stiffness: stiff, damping: damp})
              };
    });
    return endValue;
},
```

React-Motion 提供了一个非常好的 demo 页面（*http://bit.ly/2hWextu*），你可以用它来尝试所有的参数。

现在，在 `<StaggeredMotion />` 组件里，我们一开始使用的是在数组里设置的默认值，它会在组件加载完成的时候（同时也会创建原始的级联），更新指定的样式和状态值。然后，我们将 *x* 和 *y* 的值设为长宽的一半，这样可以保证触屏和鼠标单击事件能在屏幕内得到响应。还可以为了一些效果，通过在 `getStyles()` 方法里设置 `i*10` 来旋转整个圆：

```
render() {
    let arr = [], amtHalf = 175;
    for (var i = 0; i < 50; i++) {
        arr.push({x: 0, y:0, rotate:0});
    }
    return (
        <div>
            <StaggeredMotion
                defaultStyles = {arr}
                styles={this.getStyles}>
                {lines =>
                <div className="demo">
                    {lines.map(({x, y, rotate}, i) =>
                    <div key={i}
                        className={`playthings s${i}`}
                        // 需要在 CSS 里将 $amt 的值减半，这样可保证
                        // 鼠标位于生成对象的中心部位
                        style={{
                          WebkitTransform: `translate3d(${x - amtHalf}px,
                            ${y - amtHalf}px, 0) rotate(${rotate}deg)`,
                          transform: `translate3d(${x - amtHalf}px,
                            ${y - amtHalf}px, 0) rotate(${rotate}deg)`
                        }} /> )}
                </div> }
```

```
            </StaggeredMotion>
        </div>
    );
},
```

该结果非常有趣，并且有非常灵活的交互动画。它可以根据用户的操作来改变方向。

167

第 15 章

动“不可动者”：通过改变属性使用原生JavaScript实现动画

使用 JavaScript 处理动画的强大之处在于，不受 CSS 中动画概念的限制。有很多属性在改变后会产生惊人的效果，甚至很多属性我们还从来没有尝试过。

requestAnimationFrame()

制作 SVG 动画效果不需要任何库。一个优雅且高性能的制作方法是使用 `requestAnimationFrame()`（简写为 rAF）。rAF 可以替换其他原生 JavaScript 方法，比如 `setInterval()`（尽管这个方法已经通过 rAF 开创的特性来改进了）。

rAF 的工作方式是在下一次重绘浏览器时，将动画更新相关函数传递给浏览器。rAF 调用不会创建一个嵌套的调用堆栈，因为这可能导致性能问题。相反，它们将回调函数添加到由浏览器管理的队列中，并且在特定时间只有一个函数实例在运行。

`requestAnimationFrame()` 的神奇之处在于，它能以每秒 60 帧的速度运行，但在浏览器底层，它会根据你的设备情况计算出运行的速度：桌面端运行得更快，在移动设备上运行速度会变慢。当标签切换到后台时，它也会停止工作，因此这样可以节省电池寿命，同时在不需要运行的时候不会占用资源。这样处理节省了开发者的工作，我们不需要在 `setInterval()` 中手动处理这些问题，也不需要为不同的浏览器或不活动的标签设置不同的间隔时间。

rAF 的浏览器支持情况

之前如果想使用 `requestAnimationFrame()`，需要添加浏览器前缀和 polyfill，幸运的是，浏览器对此的支持已经提升，现在只有在需要支持 IE 9 或更早版本的情况下才需要这样做。Opera Mini 根本不支持 rAF，但不会在客户端运行任何 JS。它只是使用网站初始的 JS 在代理服务器上构建站点，然后将其发送给用户。请记住，使用 Opera Mini 浏览器来浏览网页，基本上类似在看一个网页的截图 / PDF。

168

rAF 的语法如下所示：

```
function animateSVG() {
    requestAnimationFrame(animateSVG);
}
requestAnimationFrame(animateSVG);
```

也可以在 IIFE（立即调用的函数表达式）中使用 rAF：

```
(function animate() {
    requestAnimationFrame(animate);
}());
```

当然，不需要将动画的所有代码都放在函数中——也可以调用另一个函数，这个函数用于动画更新。rAF 中的回调函数通常用于判断动画是否已经“完成”（除非你想要一个无限循环的动画）。

在下面的演示（参见图 15-1）中创建了一个 SVG 圆组成的对象组。前文提到过，使用 SVG 绘制圆需要三个属性：`r` 表示半径；`cx` 表示圆心在直角坐标系中 *x* 轴上的位置；`cy` 表示圆心在 *y* 轴上的位置。在此演示（*http://bit.ly/2hL5UFs*）中将使用 `cx` 和 `cy` 的值来更新圆的位置，而这两个属性在 CSS 中不是用于动画的。

在这个例子中首先声明变量，并将属性赋给变量。使用 `.innerWidth` 和 `.innerHeight`（获取页面的高度和宽度）将宽度和高度设置为页面的大小，并使用这些值来创建我们的 `viewBox`。之后定义了一些变量来代表重力、摩擦力和一个空的物体以供以后使用：

169

```
var svg = document.createElementNS("http://www.w3.org/2000/svg", "svg"),
    svgNS = svg.namespaceURI,
    vbx = document.createElementNS(svgNS, "viewBox"),
    width = window.innerWidth,
    height = window.innerHeight,
    gravity = 0.00009,
    friction = 0.000001,
    lots = [],
    prevTime;
```

```
document.body.appendChild(svg);
document.body.style.background = '#222';
svg.setAttribute("viewBox", "0 0 " + width + " " + height);
svg.setAttribute("width", width);
svg.setAttribute("height", height);
```

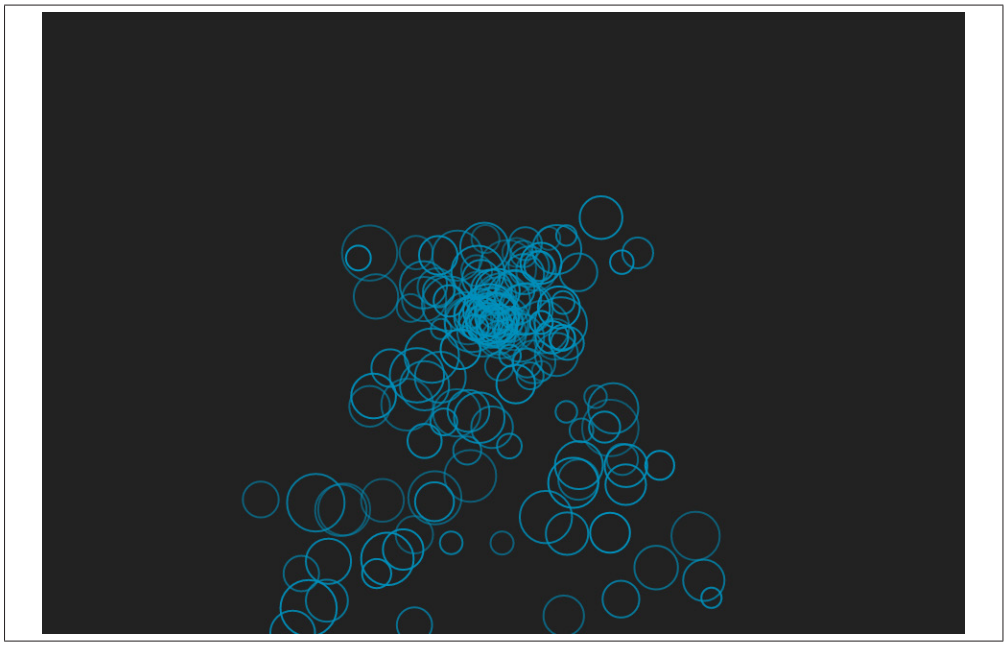


图15-1：一个用对象组创建的粒子喷泉效果，它使用requestAnimationFrame()来更新SVG属性

正确的命名空间

因为SVG需要写在不同的命名空间中，所以创建元素有点复杂，需要使用`document.createElementNS()`来创建一个SVG命名空间的URL，之后可以随时通过任何现有SVG元素的`.namespaceURI`属性访问这个URL。通常我会如以上代码所示，创建一个变量`svgNS = svg.namespaceURI`，以便缓存并快速引用它。

现在可以使用一个构造函数来创建一个气泡，`new Bubble()`，它定义了这个对象的初始状态。这个函数通过两个参数：不透明度和半径来随机创建气泡。之后把气泡添加到SVG的`viewBox`中间：

```
function Bubble(opacity, radius) {
    this.init(width/2, height/2, 0, 0);
    this.opacity = opacity;
```

```
    this.radius = radius;
    var circ = document.createElementNS(svgNS, "circle");
    svg.appendChild(circ);
    circ.setAttribute("r", this.radius);
    circ.setAttribute("fill", "none");
    circ.setAttribute("stroke-width", "1px");
    this.circ = circ;
}
```

之后创建对象的原型，为所有 Bubble 对象定义两个共享方法——用于设置初始位置和速度的 init() 方法，以及通过摩擦力（减慢速度）和重力，计算加速度来更新气泡位置的 update() 方法（相关的物理公式附在了注释中）。

```
Bubble.prototype = {
    init: function (x, y, vx, vy) {
        this.x = x;
        this.y = y;
        this.vx = vx;
        this.vy = vy;
    },
    update: function (dt) {
        var acceleration = -Math.sign(this.vx) * friction;
        // distance = velocity * time + 0.5 * acceleration * (time ^ 2)
        this.x += this.vx * dt + 0.5 * acceleration * (dt ^ 2);
        this.y += this.vy * dt + 0.5 * gravity * (dt ^ 2);
        // velocity = velocity + acceleration * time
        this.vy += gravity * dt;
        this.vx += acceleration * dt;
        this.circ.setAttribute("cx", this.x);
        this.circ.setAttribute("cy", this.y);
        this.circ.setAttribute("stroke", "rgba(1,146,190," + this.opacity + ")");
    }
};
```

每个气泡会调用 init() 函数两次。第一次创建气泡并保持居中和静止，之后用独立的方法使气泡的速度随机化。通过不透明度和半径参数来初始化一个气泡，并将它们 push 到之前创建的数组中（我称之为 lots，有时很难命名——你可以在 Twitter 上跟我讨论你喜欢的命名，但说句实话，我可能会发送一只猫皱眉头的 GIF 图作为回应）：

171

```
for (var i = 0; i < 150; i++) {
    setTimeout(function () {
        var single = new Bubble(0.5+Math.random()*0.5, 5 + Math.random()*10);
        initBubble(single);
        lots.push(single);
```

```
    }, i*18);
}

...

function initBubble(single) {
    single.init(width/2, height/2, -0.05 + Math.random()*0.1, -0.1 + Math.
random() * 0.1);
}
```

之后使用 requestAnimationFrame() 循环遍历 Bubbles 数组（lots）并激活它，使用在 Bubble对象原型中定义的 update() 方法来重新绘制位置。下面的代码做了一些小的调整，当页面的高度很小时，可以让气泡在更新之前一直下降，其他情况下才频繁地更新气泡位置：

```
(function animate(currentTime) {
    var dt;
    requestAnimationFrame(animate);
    if (!prevTime) {
        prevTime = currentTime;
        return;
    } else {
        dt = Math.min(currentTime - prevTime, 25);
        prevTime = currentTime;
    }
    for (var i = 0; i < lots.length; i++) {
        lots[i].update(dt);
        if (height < 500) {
            if (lots[i].y > height) {
              initBubble(lots[i]);
            }
        } else {
            if (lots[i].y > height*0.85) {
              initBubble(lots[i]);
            }
        }
    }
}());
```

172

通过 requestAnimationFrame()，不需要添加额外的资源便可以灵活地构建动画，也可以使用那些并不标准的动画属性。事实上，大多数 JavaScript 动画库都使用 rAF，所以如果在不抽象（使用封装好的库）的情况下更加理解动画，使用 rAF 构建是一个很好的方法。

请记住，一些库提供的抽象对于保持代码整洁、清晰是很有用的，因此在生产环境中，

使用库可能更有意义，但每个站点都是不同的，可能具有不同的要求。

GreenSock 的 AttrPlugin

GreenSock 的 AttrPlugin 包含在 TweenMax 中（参见第 8 章），因此不需要在 *TweenMax.min.js* 之外添加任何其他资源。 `attr` 的语法与上文的动画属性略有不同：

```
TweenMax.to(".trial", 3, {
    attr: {
        d: "M 100 300 C 125 200 175 200 200 100 Q 250 550 300 300
        Q 350 50 400 450 C 450 550 450 50 500 300
        C 550 50 550 550 600 200 A 50 50 0 1 1 700 300"
    },
    ease: Expo.easeOut
});
```

从以上代码中可以看到如何将 `d` 属性放在 `attr` 对象中，以与其他属性不同。对于其他可动画（animatable）属性，我们将它们用在 `attr` 对象之外：

```
TweenMax.to(".trial", 3, {
    attr: {
      d: "M 100 300 C 125 200 175 200 200 100 Q 250 550 300 300 Q 350 50 400 450 C
      450 550 450 50 500 300 C 550 50 550 550 600 200 A 50 50 0 1 1 700 300"
    },
    x: 300,
    ease: Expo.easeOut
});
```

从以上代码中可以看到实现了一个 `path` 属性的补间动画。如果存在相同数量的路径点， 173
这很容易做到，不一定需要 MorphSVG（参见第 10 章）。事实上，如果花时间了解一下 `path` 属性，可以通过补间 `path` 的值来创建一些相当复杂的路径效果。

也可以对其他属性创建补间动画。例如，可以使用下面的 JavaScript 创建一个动态的渐变蒙版（*http://bit.ly/2h77stW*）：

```
TweenMax.fromTo("#stop1", 1.5, {
    attr:{offset:-1}
}, {
    attr:{offset:1},
    repeat: -1,
    yoyo: true,
    ease:Linear.easeNone});
```

```
TweenMax.fromTo("#stop2", 1.5, {
    attr:{offset:0}
}, {
    attr:{offset:2},
    repeat: -1,
    yoyo: true,
    ease:Linear.easeNone});
```

SVG 代码如下：

```
<svg width="500"
     height="200"
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">
<defs>
     <linearGradient id="Gradient">
        <stop id="stop1" offset="0" stop-color="white"
           stop-opacity="0" />
        <stop id="stop2" offset="0.3" stop-color="white"
           stop-opacity="1" />
        </linearGradient>
        <mask id="Mask">
            <rect x="0" y="0" width="500" height="200"
               fill="url(#Gradient)"  />
        </mask>
    </defs>

    <rect x="0" y="0" width="500" height="200" fill="#480048" />
    <rect x="0" y="0" width="500" height="200" fill="#C04848"
      mask="url(#Mask)" />
</svg>
```

以上代码本质上创建了一个蒙版，并使用渐变来定义其不透明度，之后对 `offset` 属性进行补间动画，从而创建了一个高性能的渐变动画效果。如果将此与背景渐变动画进行比较，其效果会更好，因为这个动画造成的重绘较少。

174 也可以使用滤镜进行动画处理，比如通过 `stdDeviation`（高斯模糊）滤镜。在下面的代码中，使用 MorphSVG 来遍历所有路径点，并通过更新 `stdDeviation` 滤镜来产生模糊效果，以创建一个起伏的火焰效果，这个效果很自然（参见图 15-2）。

图15-2：如果你这样使用动画，可以得到不稳定的运动效果，这样对于元素来讲非常自然

以下是相关的代码：

```
function flame() {
    var tl = new TimelineMax();

    tl.add("begin");
    tl.fromTo(blurNode, 2.5, {
        attr: {
            stdDeviation: 9
        }
    }, {
        attr: {
            stdDeviation: 3
        }
    }, "begin");
    var num = 9;
    for (var i = 1; i <= num; i++) {
        tl.to(fStable, 1, {
            morphSVG: {
                shape: "#f" + i
            },
            opacity: ((Math.random() * 0.7) + 0.7),
            ease: Linear.easeNone
        }, "begin+=" + i);
    }
    ...
```

175

```
    return tl;
}
```

实际应用：viewBox 动画

在页面中使用数据可视化是传达信息的有力方法。前面，我们已经讨论过如何使用它来隐藏和显示响应式开发的信息（*http://bit.ly/2mHuDKn*）。使用 SVG 时可以通过使用 `viewBox` 像摄像机改变焦距一样隔离页面上的相关信息，突出观看者想看的信息。这种技术有很多用途，我们将会考虑一种新的动态处理方式。

在开始对视窗进行动画之前会讨论 SVG 中的 `viewBox` 概念。在这里会介绍一些基础知识，如果想深入了解，还有一些很棒的文章（*http://bit.ly/2mpCcJo*）可以帮助你（*http://bit.ly/2mB09KS*）。

`viewBox` 是你在 SVG 中看到的可见区域大小。它由 4 个坐标值定义：`min-x`、`min-y`、`width` 和 `height`，如图 15-3 所示。

图15-3: 一个完整的SVG实例

如图，周围的黑色框定义了 `viewBox`。如果熟悉 Illustrator 的人会知道这便是“画板”。
176 在 Illustrator 中可以通过“文件→文档设置→编辑画板”来更改 Illustrator 中的画板区域。之后可以裁剪图像并更改可见区域。如果确定图形大小就是想要的 `viewBox` 大小，可以用“对象→画板→适合图稿边界”来快速完成。

可以改变 SVG 的宽度和高度（*http://bit.ly/2hKDvRl*），同时保持 `viewBox` 不变。

可以像 SVG DOM 一样沿着一个网格来绘制它。该网格可以缩小和放大，但是网格的宽高比保持一致。在图 15-4 中，我们将 SVG 绘制在网格 *x* 轴和 *y* 轴的原点。宽度为 384.5，高度为 250。

图15-4：原始视窗的坐标

如果选择图片中的那些房屋并合为一组，可以看到它们是如何定位在图上的，如图 15-5 所示。

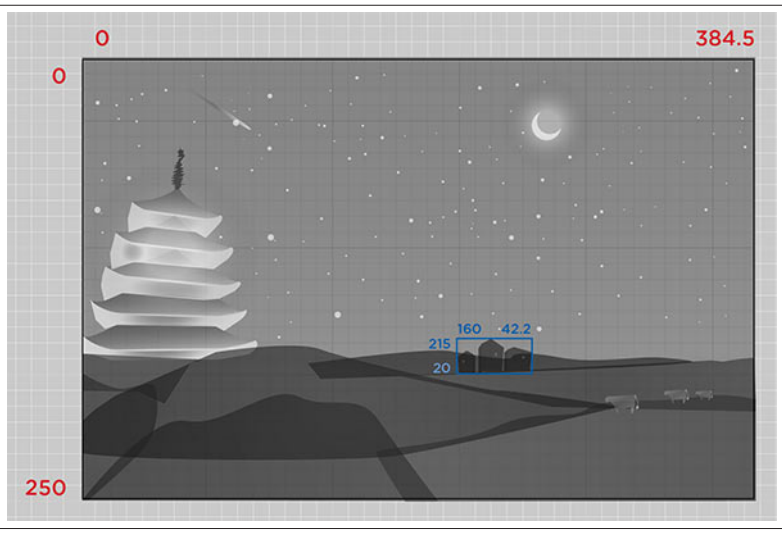


图15-5：SVG中的坐标，以及图形中viewBox里的一组元素

177 通过将 viewBox 更改为 "215 160 42.2 20"，我们可以将整个可见区域裁剪为房屋。

可以通过手工测量来找到这个 viewBox 坐标，但这样很麻烦，因为视窗本身是可扩展的。幸运的是，我们可以使用一个原生方法：调用 getBBox()（*https://www.w3.org/TR/SVG/types.html*）。调用该方法时会返回元素的边界框，并且不包括填充、蒙版或滤镜效果。调用时返回一个 SVGRect 对象（*http://bit.ly/2mpC6lf*），但它不可见。

很酷的事情是，SVGRect 对象会包含 4 个属性：x-min、y-min、width 和 height（参见图 15-6），这样操作起来就变得很简单了。

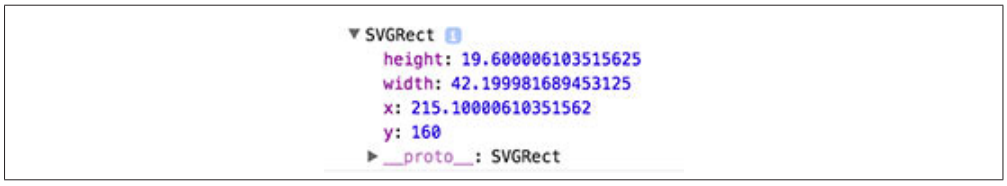


图15-6：一个 SVGRect 对象

这对我们来说非常方便，因为这意味着要动态更新 viewBox，只需将对象中的值用一个字符串存储起来，如下所示：

```
var newView = "" + hb.x + " " + hb.y + " " + hb.width + " " + hb.height;
```

178 如果使用 ES6 模板语法，则可以使其更加清晰：

```
const newView = `${hb.x} ${hb.y} ${hb.width} ${hb.height}`;
```

可以直接将 SVG 的 viewBox 属性设置为这个字符串：

```
const houses = document.getElementById("houses");
const hb = houses.getBBox();

console.log(hb);

const newView = `${hb.x} ${hb.y} ${hb.width} ${hb.height}`;

const foo = document.getElementById("foo");
foo.setAttribute("viewBox", newView);
```

现在大功告成了（参见图 15-7）。

图15-7：通过SVGRect对象来更新viewBox后的效果

动态改变 viewBox 值有几种方法，均使用 JavaScript（暂时）：

- 可以用 polyfill 来使用 `requestAnimationFrame()`（*http://bit.ly/2lkN4nc*）随时间变化以更新 viewBox 的坐标值。
- 可以使用 GreenSock 的 AttrPlugin（与 TweenMax 捆绑在一起）来动态改变 viewBox 的值。
- 可以使用 React-Motion 来更新单击事件的属性。

GreenSock 很赞的一点是它可以为任何两个整数设置补间动画，因此 GreenSock 非常好用。以下示例因为会包含其他动画，为了快速完成效果，我们会使用 GreenSock，使用其他方法也可以。 179

SVGRect 返回一个矩形

要注意，即使不是一个元素或元素被变换后并没有对角线，SVGRect 也永远返回一个矩形。获取一个圆的边界框时，会首先获取原始坐标进行转换，然后在父坐标系中找到该对角线所在的边框，这就是为什么生成的边界框矩形有时比 SVG 形状大得多。

图 15-8 及其对应的演示（*http://codepen.io/sdras/pen/MwxRBL*）里包含了一些旋转 / 变换的图形，以及带有边框的图形，这可以帮助理解边界框的概念。

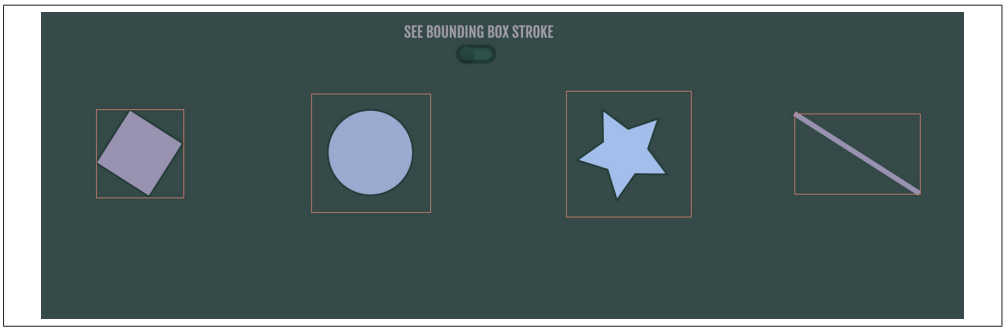


图15-8：当一个SVG元素被变换时，边界框不会随之改变——跟对角线无关;即使图形是圆和线，边界框也会展开并创建一个矩形

180 在 *http://codepen.io/sdras/pen/dXoLEJ* 上有一张地图，当用户与之进行交互时，会展示他们选择的具体国家的信息。

181 我在热点上添加了重复运行的动画，还在元素上使用了一些简单的 data 属性，以便可以存储和使用这些信息来进行动画处理。一致的命名在这里是很重要的。下面的代码展示了如何控制某国家的扩大和显示：

```
<g data-name="usa" class="hotspot">
    <circle id="dot2" cx="221" cy="249" r="2.4" fill="url(#radial-gradient)"/>
    <circle id="dot1" cx="221" cy="249" r="2.4" fill="url(#radial-gradient)"/>
    <circle id="dotmid" cx="221" cy="249" r="2.3" fill="#45c6db"/>
</g>
```

热点元素还添加了一些额外的填充区域，以便在移动设备和触控设备中可单击区域将其变得足够大：

```
.hotspot {
    cursor: pointer;
    padding: 20px;
}
```

之后编写一个函数，单击后，传入 data 属性，并根据国家的形状更新 viewBox。宽度增加了 200，以便于容纳国家旁边的文本：

```
// 交互
function zoomIn(country) {
// 局部放大
var currentCountry = document.getElementById(country),
    s = currentCountry.getBBox(),
    newView = "" + s.x + " " + s.y + " " + (s.width + 200) + " " + s.height,
```

```
    group1 = [".text-" + country, ".x-out"],
    tl = new TimelineMax();

    tl.add("zIn");
    tl.fromTo(map, 1.5, {
        attr: { viewBox: "0 0 1795.2 875.1"}
    }, {
        attr: { viewBox: newView }
    }, "zIn");
    tl.to(".text-" + country, 0.1, {
        display: "block"
    }, "zIn");
    tl.fromTo(group2, 0.25, {
        opacity: 1
    }, {
        opacity: 0,
        ease: Circ.easeIn
    }, "zIn");
    tl.fromTo(currentCountry, 0.35, {
        opacity: 0
    }, {
        opacity: 1,
        ease: Circ.easeOut
    }, "zIn+=0.5");
    tl.fromTo(group1, 0.5, {
        opacity: 0
    }, {
        opacity: 0.65,
        ease: Sine.easeOut
    }, "zIn+=1");
}

$(".hotspot").on("click", function() {
    var area = this.getAttribute('data-name');
    $(".x-out").attr("data-info", area);
    zoomIn(area);
});
```

182

如果想减少代码量，可以把时间轴 `tl` 也封装在一个函数中，当有人单击 × 时就可以反向执行动画，但是尝试之后发现动画比预想中有偏移，所以仅仅做了一层封装来精简一些代码。也可以用 `tl.to` 代替 `tl.fromto`，不过重新运行动画时，提供初始值可以增强代码的可维护性（特别是当你不知道谁可能会更新你的代码时）。

本例使用了 jQuery 而不是普通的 JavaScript。

CSS 中的 viewBox

已经有一个让 `viewBox` 成为 CSS 属性的提议，我对此非常支持。如果你也想支持它，可以提出一些技术性反馈，或者在现有的讨论中为你支持的观点点赞支持。控制 `viewBox` 的 CSS 属性是非常有用的，可以很容易据此来使用媒体查询和动画，甚至可以减少这些更新来触发的布局重绘和重排。

另一个演示：一个有引导作用的信息图

在另一个 demo 中（*http://codepen.io/sdras/full/VjvGJM/*），包含一个小流程图来展示如何使用该技术指导用户。这个特殊的图可引导用户选择合适的图像格式（参见图 15-9）。

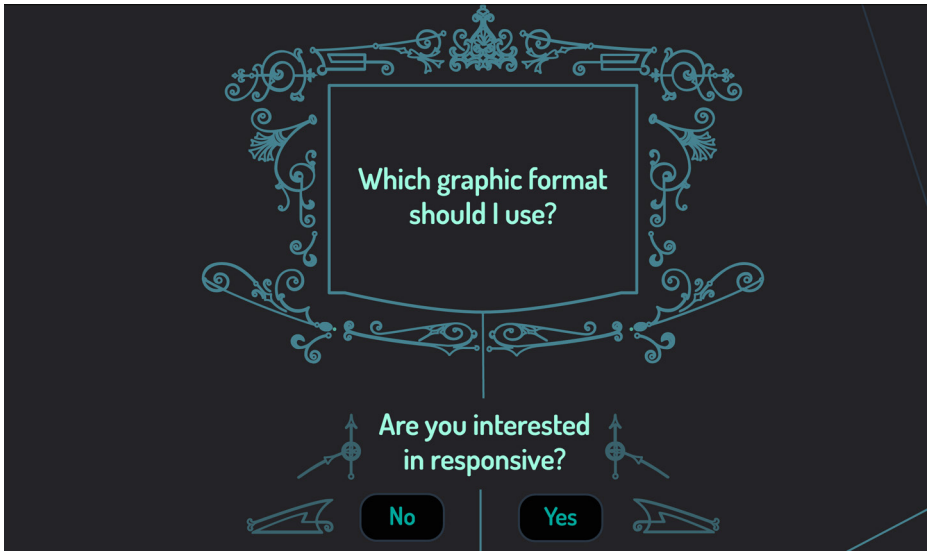


图15-9：使用viewBox和getBBox()动态更新的动画版流程图

183 **注意** 动画可能导致眩晕，如果你有类似“前庭障碍”这样的情况，不要观看这个动画。如果这个动画嵌入到一个线上站点，可能需要提供一个开关选项来让用户选择是否观看。

第 16 章 185

响应式动画

Web 动画是非常微妙的，因为我们必须根据带宽、代码兼容性和产品设计来调整我们的动画设计。在这一章中，我们将讨论如何创建响应式（可伸缩）动画的技术，还将介绍如何使用创建动画的不同方式来实现在众多设备中有一个较好的用户体验。

快速响应的技巧

我们至少应该确保在移动端创建的动画能发挥很好的作用，而且在移动设备上创建的互动能提供所有功能，这里可以使用像 ZingTouch（*http://bit.ly/2m9t2Av*）或 Hammer（*http://hammerjs.github.io/*）这样的库，使用滑动或手势检测来通过本机的交互检测。

Web 响应式设计可以在 `<meta>` 标签中指定 `initial-scale=1.0`，以便在调用操作之前，设备不需要等待默认的 300ms。互动触摸事件必须有一个更大的触摸目标（40×40px 或更大）或者使用 `@media(pointer:coarse)`。

GreenSock 和响应式 SVG

使用 GSAP 的首要原因是其对 SVG Transform 属性的跨浏览器支持。SVG 中的旋转非常麻烦（*http://bit.ly/2lktYgW*）。在几乎所有的浏览器中，`transform-origin` 都存在一定的问题，并且在 IE 中完全不受支持。当试图以一种响应式方式使用 Transform 属性时，这个问题变得更为严重，那是因为任何 `transform-origin` 都会被异常放大，而且难以覆盖。

GreenSock 不仅纠正了这种行为，而且还支持 IE 9，它提供了很多工具，使得用响应式 186
设计和开发变得特别稳定。目前，使用原生的 SVG 元素渲染技术，SVG Transform 属性不支持基于百分比的渲染。GSAP 通过矩阵计算解决了这个问题。

首先应该先从 SVG 本身移除 `width` 和 `height` 的属性值，定义 SVG 的 `viewBox`，然后

使用 CSS 或 JavaScript 来控制 SVG 的宽度和高度，这样可以轻松地使 SVG 适应任何终端设备的呈现。你还可以添加 `preserveAspectRation="xMinYmin meet"` 来确保所有 corre-sponding 维度都能适当地伸缩，这是默认的，但不是必需的。如果你想获得更多 viewBox 的相关知识，可以阅读 Sara Soueidan 整理的教程（*http://bit.ly/2lNbuJv*）。

GSAP 对于 SVG 还有另外三个优点，它们使 Transform 属性在 SVG 中可以更好地被使用。首先，除了 `transform-origin`，GSAP 现在已经内置了对 `svgOrigin` 的支持。这意味着你可以根据元素本身来选择是否要转换元素（在自己的轴上旋转）或在 SVG 的 viewBox 中使用坐标。使用 `svgOrigin`，你要根据 `viewBox` 坐标声明值。换句话说，如果你的 viewBox 是 `"0 0 400 400"`，并且你想围绕 SVG 的中心旋转，需要声明 `svgOrigin:"200 200"`。通常情况下你会发现移动和调整 `transform-origin` 已经足够了。在图 16-1 中，我做了一个月球旋转的动效（*http://codepen.io/sdras/pen/doZReX*），它也是 viewBox 的一部分，我使用了一个 svgOrigin 进行协调，这个动画具备响应式效果。

```
TweenMax.set(cow, {
    svgOrigin:"321.05, 323.3",
    rotation:50
});
```

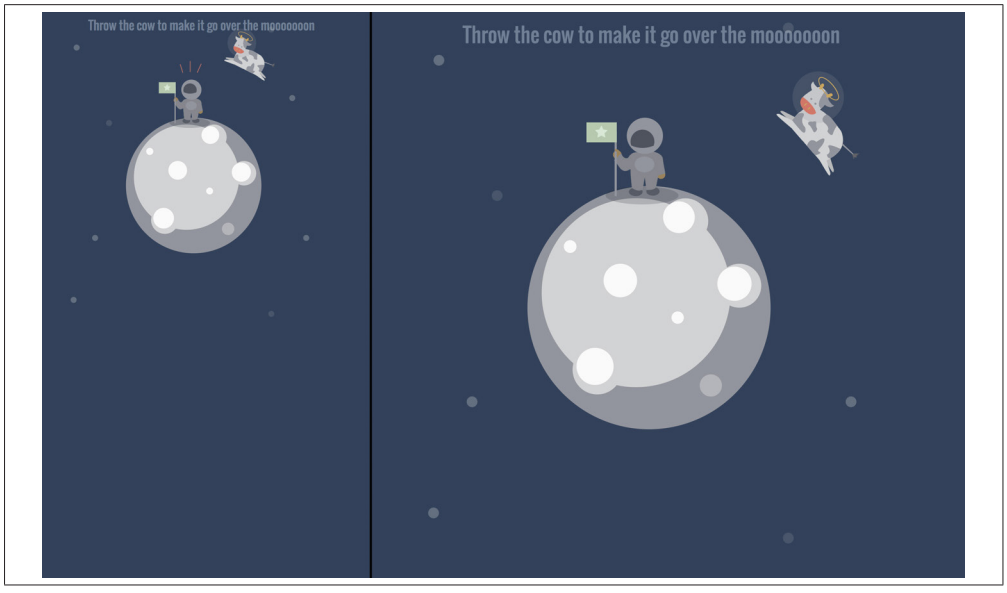


图16-1：SVG动画是可以拖放的，并可在SVG viewBox中用svgOrigin在一个点上进行旋转，因此在响应式设计中它是完全稳定的

187 接下来继续讨论 SVG 中的另一个重要特性，那就是关于 SVG 元素的 `smoothOrigin`。

通常情况下，`transform-origin`就可以改变元素的转换原点，那么这个动画就会变得复杂和违反直觉，图 16-2 所示的是一个响应式的案例（*http://codepen.io/1Marc/full/DCvFm*）。

图16-2：这个案例展示了堆叠行为，解释了transform违反直觉的行为（此案例由Marc Grabinski提供）

188 正如 Carl Schooff 录制的一段视频（*http://bit.ly/2mpuEXo*）中解释的那样，GreenSock 对 `smoothOrigin` 这个问题进行了修正。它确保更改了 SVG 元素的 `transform-origin`，
189 随后再次移动它时，不会引起任何奇怪的跳动。

当处理一个更复杂的响应式动画时，这解决了大量反直觉以及一些令人不爽的行为。GSAP 还让你在 Edge 中使用一行代码 `CSSPlugin.defaultSmoothOrigin=false` 就可以呈现原生的渲染效果。

在 GSAP 中，复杂的响应式动画的最后一个重要功能是能够让 SVG 元素基于百分比实现动画。CSS 和 SMIL 对这种类型的行为还没有很好的支持。和 BezierPlugin 一样，GSAP 为 SVG 的 Transform 提供了向后兼容和跨浏览器的支持。图 16-3 所示的就是一个 GreenSock 的动画效果：

```
var playBtn = document.getElementById("play"),
    tl = new TimelineMax({repeat:1, yoyo:true, paused:true});

tl.staggerTo(".box", 0.5, {x:"100%"}, 0.4)

play.onclick = function() {
    tl.restart();
}
```

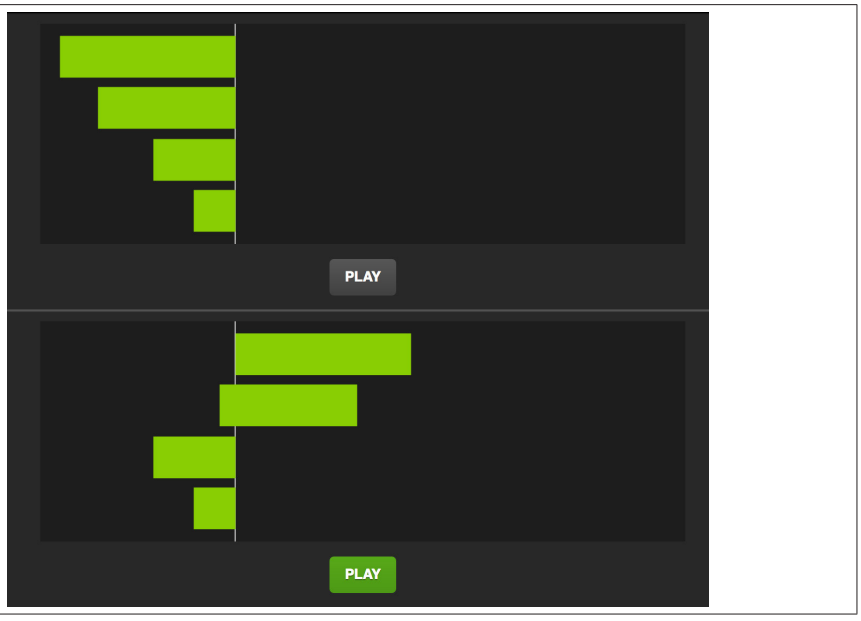


图16-3：GreenSock允许对SVG元素进行基于百分比的变换，这是响应式动画开发的一个非常好的特性

不使用 GreenSock 实现响应式 SVG

190

SVG 元素基于百分比的变换令人印象深刻且非常有用。在响应式开发中，我们充分利用了 Flexbox、百分比，并允许我们扩展到 fit 容器。但是，当你移动到 SVG 时，令人惊奇的是，你可能不需要它们。SVG 的变换是在 SVG 画布上实现的，不是浏览器窗口定义的绝对像素值。我们根据 SVG DOM 移动元素，元素并不是唯一可伸缩的东西，所以响应式的变换和动画也是一样的。

不需要依赖任何媒体查询，就可以根据 *x* 轴和 *y* 轴移动元素，比如：

```
tl.staggerFromTo($iPop, 0.3, {
    scale: 0,
    x: 0,
    y: 0
    },
{
    scale: 1,
    x: 30,
    y: -30,
    ease: Back.easeOut
}, 0.1, "start+=0.3")
```

图 16-4 所示的就是对应案例（*http://codepen.io/sdras/full/jPLgQM/*）的演示效果。

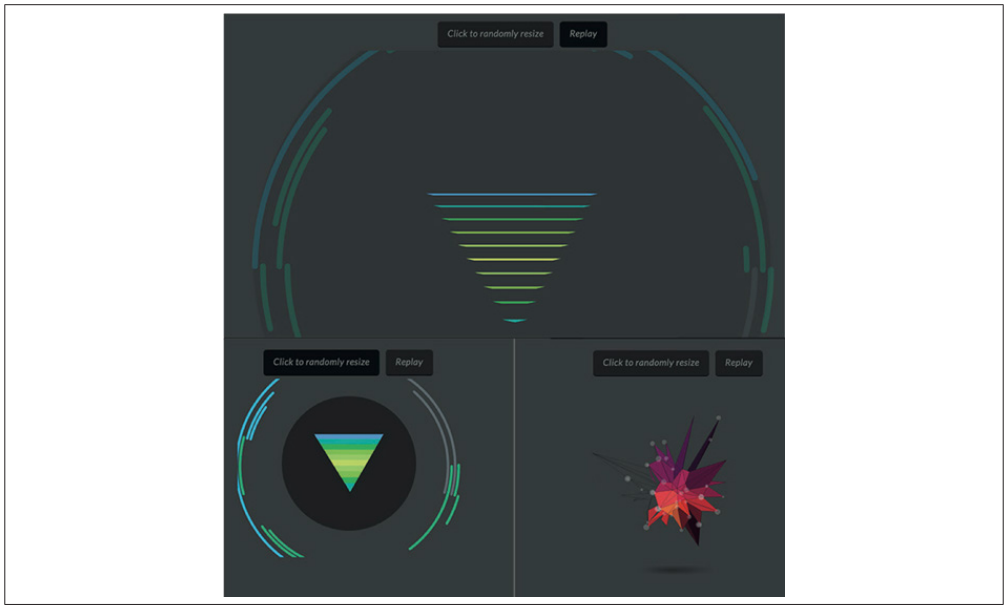


图16-4：当单击按钮时，动画会改变大小，但动画效果保持不变

191 请注意，我们这里没有使用基于百分比的变换。动画是建立在 `viewBox` 基础上的行为，因此，响应式开发变得像删除宽度和高度一样简单。

仅仅“squish”一种动画就可以适应我们的视窗，而且效果很好，但是我们都知道，真正的响应式开发是一个更为复杂的过程。让我从头到尾使用工具来进行一些响应式的开发吧。

有几种方法可以做到这一点。其中的一种方法就是使用大型 SVG Sprite，并使用媒体查询来处理 `viewBox`（*http://bit.ly/2lYYYH3*）。

通过更新 viewBox 实现响应式

还记得过去人们使用信息图表的时候吗？信息图表因其受到转换的影响而变得越来越没有人喜欢用。但在用户方面，用户可以快速理解信息。它们是五颜六色的，被创建时
192 就能很清楚地向用户展示信息的对比。图形对公司知名度和品牌知名度的影响是惊人的（*http://bit.ly/2mDH3Tq*）。Brain 的博客（*http://bit.ly/2lN4nkk*）中有大量这方面的数据：

- 网站流量增加了 400% 以上。
- 增加了将近 4500% 可访问区。
- 新网站访问者数量增加了近 78%。

但是所有这些帖子都有一个共同点，那就是它们至少都有两年以上了。如果信息图表有这种潜在的性能，为什么它们看起来被认为是一个过时的潮流呢？

手机可能是一个引爆点（*http://bit.ly/2m3Jth6*）。在桌面电脑上令人兴奋和无所不包的信息图片在移动端成了一件费力不讨好的事情。

另一个原因可能要简单一些。当一个概念不适应当前的文本，它就会消失。在 Web 上有更多的交互性和动态元素，静态图形与视觉上更令人兴奋的东西没有可比性。动效胜过一切。

图 16-5 所示的是对应的示例效果（*http://codepen.io/sdras/full/JdJgrB/*）。请记住，文本转换的进度是为了演示动画而不是内容。

图16-5：使用viewBox实现用于移动端和桌面的信息图表

在设计方面，传统的信息图表通常使用一种沙龙（Salon）风格作为视觉上的加载方法。 193
在这里，我们仍然填充了可用的图像区域，但是新的设计使其更干净。这里没有过多地使用元素，因为与传统的静态信息图表不同，在移动端有过多的元素会使用户迷失方向，而且也会令其加载更多的东西，性能会受到一定的影响。

在响应式设计方面，整个动画演示过程都是流畅的，将信息图表放在一个断点上，然后将元素移动到不同的位置。这样，SVG 的主元素就可以在这个节点上流畅地响应了。尽管我们首先设计了 PC 桌面端，但是媒体查询的原则是移动端先行。使用 SVG 可以很容易地让动画流畅运行，通过 JavaScript 在移动端上调整 `viewport` 即可实现：

```
var shape = document.getElementById("svg");                    194

    // 媒体查询事件处理器
    if (matchMedia) {
```

```
        var mq = window.matchMedia("(min-width: 826px)");
        mq.addListener(WidthChange);
        WidthChange(mq);
    }
    function WidthChange(mq) {
        if (mq.matches) {
            shape.setAttribute("viewBox", "0 0 765 587");
        } else {
            shape.setAttribute("viewBox", "0 0 592 588");
    }
};
```

接下来使用 GreenSock 来对动画做处理，利用时间轴和擦除动画来找到不同的时间点来与滑动条进行交互。下面是时间轴上图形显示中的一个信息示例。请注意，我们已经为所有动画添加了一个相对的时间以使用一个标签：

```
tl.add("likely");
tl.to($(".p1"), 0.3, {
    scale: 1.3,
    transformOrigin: "50% 100%",
    fill: $blue,
    ease: Bounce.easeOut
    },"likely")
    .to($effect, 0.3, {
        y: -10,
        ease: Circ.easeOut
    }, "likely")
    .to($eLine, 0.3, {
        stroke: $orange,
        ease: Sine.easeOut
        }, "likely")
    .fromTo($(".d1"), 0.3, {
        opacity: 0,
        scale: 0.7
    }, {
        opacity: 1,
        scale: 1,
        ease: Back.easeOut
    }, "likely")
    .to($m1, 0.3, {
        fill: $green,
        ease: Circ.easeOut
    }, "likely");
```

195 可以通过添加一个 <title> 元素来提高图形的可访问性。还可以在 <svg> 元素中提供一

个 aria-labelledby 属性，以加强这两个元素之间的关系：

```
<svg aria-labelledby="title" id="svg" xmlns="http://www.w3.org/2000/svg"
        viewBox="0 0 765 587">
    <title id="title" lang="en">Circle of icons that illustrate Global Warming
    Solutions</title>
```

如果需要，可以为 SVG DOM 中的任何元素提供 title。你可以在 Dudley Storey 的文章（*http://bit.ly/2lrOtZR*）中了解到更多关于这方面的信息。在这个示例中，将文本分隔开，这样用户可以在屏幕上更清晰地阅读想要的信息。这是对原始信息图形的改进。

这个演示仅仅是一个草图，是用来思考的方法。通过它们可以给可共享的信息提供更多的用处和响应式的动画。同样的事情也可以通过 PNG、CSS、canvas 和其他各种方法来实现。现在在 Web 上支持的工具足以令人感到兴奋，它们可以为旧的概念注入新的生命。

具有多个 SVG 和媒体查询的响应式

我们在第 3 章中详细介绍了一个解决方案。另一种方法是使用联锁部件来设计我们的动画，就像俄罗斯方块一样，并使用多个可以重新配置的 SVG。让我们看看后者，示例（*http://codepen.io/sdras/full/waXKPw/*）如图 16-6 所示。

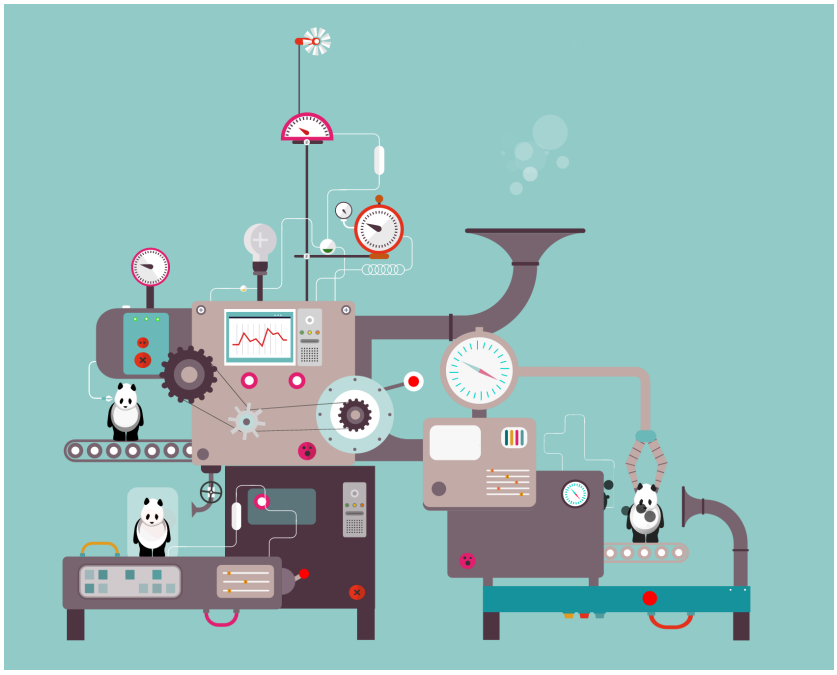


图16-6: 一个富交互、充满活力的激光熊猫加工厂

196 在激光熊猫加工厂（Huggy Laser Panda Factory）的这个示例中有三个不同的部件。为了保证代码有组织，每个部分都可以接受一种类型的用户交互，然后触发它自己的时间轴（参见图 16-7）。

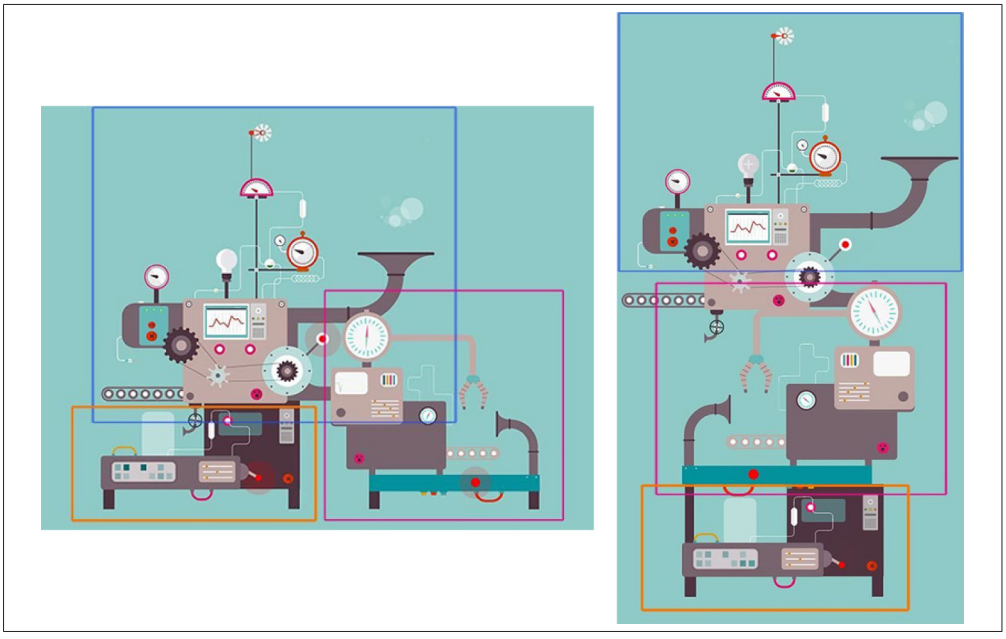


图16-7：每一个SVG的联锁和重新配置，这取决于视窗尺寸的大小

197 保持内联的 SVG 不同于其他，也使我们可以折叠它们，并根据可变的视窗的百分比或整数移动它们，使动画在移动端和未来的迭代中变得更为灵活。我们已经为桌面端程序设计了一个初始视图，并可为较小的显示进行重新配置，包括 `transform: scaleX(-1)`。

```
@media (max-width: 730px) {
    .second {
        width: 70%;
        top: -90px;
        margin-left: 70px;
        transform: scaleX(-1);
    }
}

@media (min-width: 731px) {
    .second {
        width: 350px;
        margin-left: 365px;
```

```
        top: 370px !important;
    }
}
```

每个构建块都有自己的函数，命名为它所服务的动画的一部分。这避免了范围的问题，也使所有的事情变得有条理。用户只能触发与该动画相同的 SVG 或构建块的行为。我们先暂停时间轴，使用按钮或组来重新启动它：

198

```
var triggerPaint = new TimelineMax({paused:true});
triggerPaint.add(paintPanda());

$("#button").on("click", function(e){
    e.preventDefault();
    triggerPaint.restart();
});
```

我们还有一个包含文档中许多元素的循环时间轴。我们在它的开头设置了一个相对的标签，这样就可以在多个对象上设置循环。这很重要，因为如果让循环在时间轴上彼此跟踪，只有第一个会被触发，它将永远运行，而第二个将会无限期地等待下去：

```
function revolve() {
    var tl = new TimelineMax();

    tl.add("begin");
    tl.to(gear, 4, {
    transformOrigin:"50% 50%",
    rotation:360,
    repeat:-1,
    ease: Linear.easeNone
}, "begin");
    tl.to(wind, 2, {
    transformOrigin:"50% 50%",
    rotation:360,
    repeat:-1,
    ease: Linear.easeNone
}, "begin");

    // ...
    return tl;
}

var repeat = new TimelineMax();
repeat.add(revolve());
```

现在，我们总共有四个时间轴：三个与每个部分相关的时间轴以及全局循环时间轴。交互和动画缩放在每一个 SVG 上，所以可以自由移动和调整它们在不同视图中的配置，并且代码直接、干净且有组织。

199

花更少的精力在移动端

还是面对现实吧，移动端的网络（特别是在欠发达国家）可能相当慢。不管是你的网站上只有几个关键的动画互动，还是一个强大的 WebGL 体验，有时候在桌面端看起来很漂亮的动画不需要扩展到移动端上。

对于一个大型 canvas 动画，或者是一个非常复杂的 SVG 动画，对于用户体验来说并不重要，有时候最好的办法是把它降级或者关掉，或换成较小的动画。

Active Theory 在网站上做了一件很漂亮的工作，如图 16-8 所示。在桌面端是一个完整的粒子动画，而在移动端是一个简单的多边形。在移动领域的互动仍然很有意义，甚至超过了我们的预期。

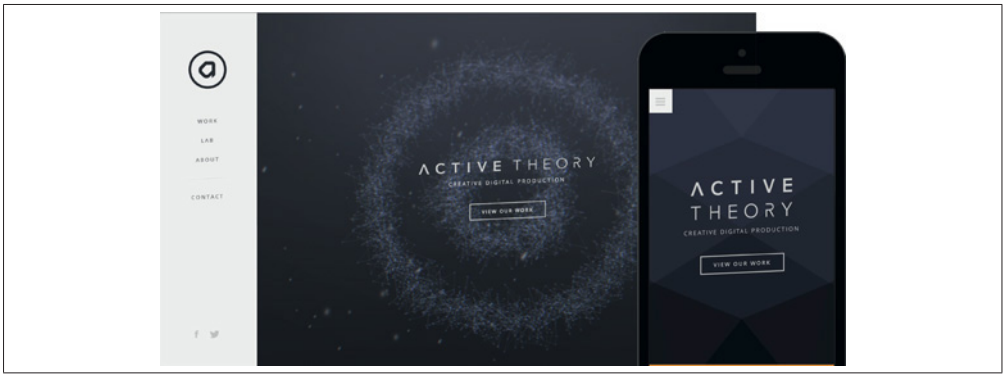


图16-8：Active Theory保持了视觉语言的一致性，在移动端上放弃了粒子动画

该团队仍然展示了它在网站上的互动能力，这一点在移动领域比动画更令人印象深刻，还节省了不少流量。

有一个计划

无论你是从设计开始到结束都考虑响应式，还是简单地把动画从移动端删除，都要有一个具体的计划。从一种设备到另一种设备的用户体验非常重要，这一点在移动领域是王者。内容、图像类型和用户带宽都应该有助于指导响应式动画的设计选择。

第 17 章 201

组件库的设计、原型化和动画原理

在现代前端工作流程中，那些能够帮助我们保持代码结构组织化、改善工作流程、降低维护成本的设计系统和组件库已经相当成熟了。如果想让这些系统运行良好并稳定地扩展，应该保证持续更新正确、有可用的技术文档和减少交流沟通上的冲突。

虽然这些系统在字体、颜色和基础架构上都有不尽人意的地方，Web 动画方面的功能也仍然是临时且杂乱无章的，但幸运的是，我们可以借助已有的架构和工作流来减少动画方面的开发阻力，创造出各个环节紧密结合且具有较好用户体验的 Web 动画。

本章，我们将内容划分为有关动画方面的设计、计划和实现三部分。

动画设计方面

Web 动画，和其他 Web 元素一样都需要经过精心设计。具体原因和细节可以参考在 Smashing Magazine（*https://www.smashingmagazine.com/2017/05/enhancing-mobile-design-ux/*）社区中发表的多篇文章：

- Amit Daliot 写的 *Functional Animation in UX Design*。
- Rachel Nabors 写的 *The State of Animation 2014*。
- Tom Waterhouse 写的 *The Guide to CSS Animation: Principles and Examples*。

作为 Web 开发者，我们的主要精力会放在排版、布局、交互和视窗变换方面。但是在实际开发过程中，还有一个需要我们关注的要素，那就是时间。

这并不是我们额外需要考虑的部分，事实上，时间越长，我们在有关排版、布局、交互和视窗变化方面需要考虑的复杂程度越多。但是，无须在这一块有太多的考虑。可以根据实际的用户体验来慢慢细化我们的 Web 动画，以创造出令人兴奋且能够引人入胜的 202

Web 动画，从而推进用户体验水平和全面优化 Web 媒介。

学会勾勒实际运动中的细节

虽然每一个人都有自己独特的工作方式，且没有任何一种方式是绝对正确的，但是，我想分享一下，我在工作过程中总结的一些关键点和工作方式。

首先，我认为应该学会观察实际物体运动的细节。你可能会笑话我，认为做到这点太简单了。 但是，我想反问一下：如果你真做到了这一点，那么请问把一杯水倒进杯子中具体需要多长时间？ 动画中怎样做到让一个人物的步态看起来清晰可辨？

想要学会这一点，我建议初学者首先从观察球的弹跳练起。由于简单，所以这个例子在一定程度上可以很好地让你明白小球的材质、重量和弹射力度。图 17-1 所示的是一个关于两个小球弹跳的对比演示（*http://codepen.io/sdras/pen/zxJWBJ*）：你能猜出哪个小球比较硬而哪个小球比较软吗？

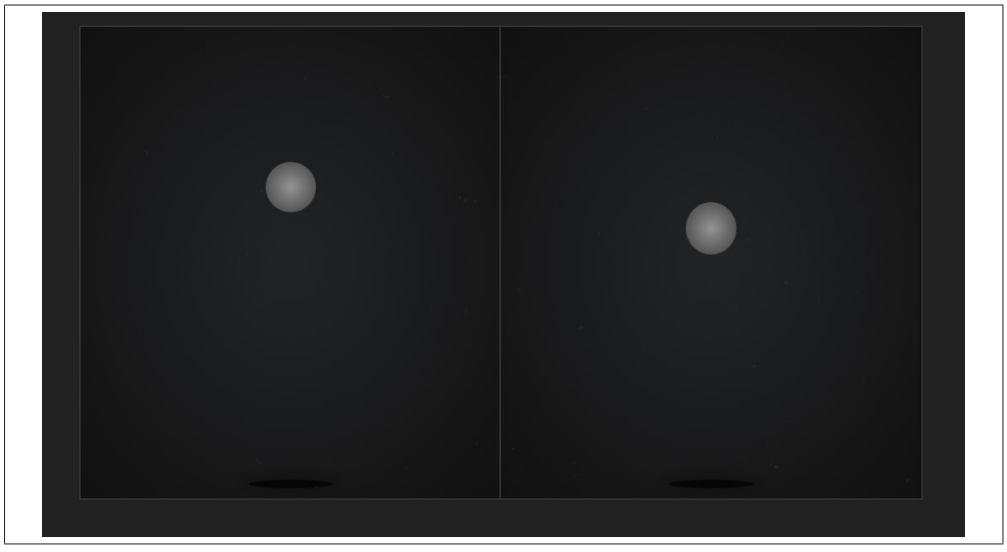


图17-1：两个小球在相同弹跳率的情况下，你可以清楚地知道两个小球在材质上的不同——哪一个小球材质比较软

怎么知道小球的材质呢？其实道理很简单，主要是观察小球的形变，左边的那个小球自始至终都是圆的，而另外一个却会因外力作用而产生形变。还有什么细节呢？左边小球运动得比较死板僵硬，而右边的小球运动起来是轻巧灵活的。尽管给它们设置的运动时间是相同的，但是它们运动过程中却包含了大量的信息。同时，为一个运动对象所设置的缓动函数也能让物体有不同的表现。

注意，尽管为它们设置了相同的运动时间，但是，它们使用了不同的缓动函数，所以，对应的动画关键帧也是不同的。这里我把上面的演示做了一些改进：只给小球描边，并且每隔一段时间闪动一下小球的形状。这样你就能看清，在某一时刻，它们在细节上的不同之处。这个演示的概念其实和早期的赛璐珞胶片动画电影很像。让我们来看看图17-2 和其对应的演示（*http://codepen.io/sdras/pen/MYdQor*）。

203

图17-2：通过显示轮廓，可以看清楚运动的“间隔”以及对比两个小球的不同

这种关注细节的思考模式也同样适用于其他次级元素。例如，如果有人搅动了玻璃瓶里的水，那么水里应该产生怎样的效果呢？如果有人踢了一块石头，那么石头在外力作用下会怎样运动？ Dribbble 网站上有一个对细节描述很好的例子（*http://bit.ly/2l8r97e*）。

就像迪士尼动画制作师 Hans Bacher 在 *Dream Worlds* 一书书中写到的，当他们在制作《美女与野兽》的时候，为了能够很好地把握细节，公司把他们直接外派到了法国和伦敦，让他们实地考察那里的古典建筑。你的公司可能不像迪士尼那样财大气粗有公派预算，但幸运的是，现在的互联网可以为你的工作提供大量关于视觉、历史和空间的信息。

合理控制动画的使用

204

不像网页中的字体、颜色等，我们通常到设计后期才会考虑为网页添加动画元素。如果把控不好会导致总体缺乏凝聚力，让网页看起来杂乱无章。如果你是一个公司的新领导不了解项目的最初规划，替换了从项目开始就使用的基础设计。那么，这种做法会引起设计师崩溃，工程停滞不前，所有人反感你所提出的建议。举一个简单的例子，你要求

公司的设计师或者技术人员制造出一套设计稿或项目原型，却不让他们用之前工作中使用的字体。这个道理在动画设计中也是一样的。请避免在项目快结束的时候，添加一些和整体设计风格不符的、看起来很华丽的动画。动画对于网页的意义从来都是在精不在多的。

那么如何合理使用动画呢？ 第一步你需要做的就是做好动画审核工作，仔细检查你的网站中所有用了动画的地方，思考这些地方是否真的适合添加动画，或者有哪些地方适合用动画，但是你却没用动画。（提示：例如表单提交的等待动画，这样可以奇迹般地提升网站的活跃度，且效果直观。）

不确定怎样做好动画审查？ Val Head 在她的 *Designing Interface Animations*（Rosenfeld Media 出版）一书中有一章专门介绍了这个问题，里面包含大量的研究实例并给出了很多良好的建议。

网上的一些优美的动画组件库所配套的文档有时也会错误地引导你去选择一些并不适合你网站的动画。记住，只有适合自己的才是最好的。举一个例子，有些动画组件库提供反转 180° 的动画或者其他一些花哨的动画组件，但是实际上我们在网站中很多时候都用不到这种动画。

下面我们会讨论两种常见的、由于没有很好地进行动画审核概念所导致的错误。

拥有特色的设计主见

很多人都会对 Material Design（*http://bit.ly/2lZ7MwV*）（Google 提出的一套设计规范，其中包含关于动画设计方面的规范）感到困惑，他们把 Material Design 看作一种动画设计规范，但是他们从来没有设计动画的经验和总结动画设计的习惯。所以当这些人每次用 Material Design 作为网站动画的设计规范时，用户看到他们的网站，都会感觉是在访问 Google 网站。

选用 Google 的设计规范而非自己的设计规范来设计网站，会让你失去一个让用户对你的网站留下深刻印象的机会。

在我看来，对于 Google 的 Material Design，重要的不是这款设计规范本身，更多的是 Material Design 作为行业内首个聚合了公司品牌的动画设计准则为我们起了示范作用：一个公司的统一动画设计规范有助于提高公司品牌形象。在网站项目设计初期，当我们开始为网站思考具体的动画实现的时候，就应考虑将公司的所有细节凝聚到统一的动画设计规范中。

205 如果你的公司是一家以值得信赖和诚信著称的保险公司，那么公司网站中所有的动画风

格都应该是比较正式的，而不能是过分华丽的。这点可以通过设置适当的缓动函数得到体现：你需要尽可能地使用线性动画而不是弹性动画或跳跃动画。但是为了让公司的品牌给用户留下更多舒适且友善的印象，你需要根据公司品牌来定制一些较复杂且生动的动画以提高公司的品牌效应。你可以参考 Zendesk（客商社交平台）或 MailChimp（世界最大的营销自动化平台）两家公司所搭建的网站。它们都是根据自己公司的品牌来定制生动的网站动画的，你还可以参考 Chris Gannon 的博客（*http://bit.ly/2lPaMK1*）（Chris Gannon 是一位 SVG 动画交互专家），Chris Gannon 制作的动画都极富魅力。特别是他提供的加载动画案例，形式看似简单但却令人兴奋，简单可依赖。

Aarron Walters 曾经在他的 *Designing for Emotion* 一书中写道："回想一下，你第一次因为某个虚拟人物的故事情节而感动流泪的情节，那很可能是动画"，他认为相较于其他事物，我们更容易记住情感方面的事。 他的书的第 7 章中总结了一些关于网站用户影响情感体验的投入产出比的数据 , 有兴趣的读者可以去读读他写的书。

有一个重要的现象我们需要注意，多数情况下，用户在浏览网页的时候只是会粗略地扫读。如果你使用 CSS 技术将一张静态的图片单纯地展现在网页中，那么用户可能一扫而过，不会在脑海里留下什么印象。而动画则不同，它允许我们将元素的展现方式化静态为动态。如果做成交互式的模式，需要单击，图片才会展现，并且是以动画的形式展现，用户的参与度会惊人地上升。

在实际开发过程中应该对动画抱有怎样的想法呢？其中有一个建议是，尽量不要尝试使用翻转动画 , 并且，尽量将你的缓动函数调整出轻滑的感觉。在这种情况下，你需要尽可能地去调整你的缓动函数，使你的动画接近自然物体滑动的感觉，并且不要在网站项目中写 `transform: scaleX(-1)`。在整个团队中，尽量不要花时间在设计翻转动画上，尽量以滑动动画的形式替代。你应该学会节省时间，避免把时间花费在关于内部样式统一的沟通上。

提升开发水平

动画作为开发过程中重要的组成部分，开发者必须了解其对应的开发流程。一般我们完成动画要按照以下几步来做：

- 专业的动画需要单独进行设计，包括：搭建实物模型、配色、情节框架设计，还有一些自己的创意和构思。
- 你的设计过程，必须遵循你的代码逻辑架构。
- 动画必须是能够为用户提供信息的。诉诸理性，并引导用户的注意力。
- 动画必须驱动公司的品牌，作为主题引导的一部分，吸引用户投入情感。

- 我们没有必要重新造轮子，动画已经早于互联网发展很多年了（可以使用之前已经造好的轮子）。

因为动画是如此吸引人，所以很容易做过度——不是屏幕内所有的东西都适合进行动画化的。你没有必要在战争一开始就用出你的秘密武器。动画可以用来作为开场或结束的象征（开场动画和结束动画），用于吸引观众的兴趣。使用动画是需要有目的性的，且需要根据用户的参与程度、绩效预算还有品牌效应来进行计划的。

Val Head 在她的 *Designing "Invisible" UI Animations* 中就谈到，一个好的动画（所处的场景）不应该显得不合适，也不应该是事后的想法。

让我们来看一个例子，Oleg Solomka（mo.js 的作者）写的泡泡布局（*http://codepen.io/sol0mka/full/yNOage/*），这个动画十分有趣，互动声音也很好听，很吸引人去参与互动。但是，用户把注意力都放在交互上了，会忽视实际需要推广的文章内容。请记住，我在上文提到的，动画的本质是以一种特殊的方式来展现需要展现的内容，而在实际环境下，一个过于吸引人的动画会转移用户的注意力，这样辛苦制作的动画起到了负面效应。

设计原型

了解关于动画设计的知识后，下一步就是付诸实际的计划。

逐步分割动画细节

当你开始制作动画之前，必须绘制一份分镜（storyBoarding）。分镜是一个很重要的过程，在这个过程中将动画进行分割，从而你可以以模块化的方式组织你的代码。在很多场景中，分镜还能够让你对动画时间进行合理分配，允许你按照步骤工作：一步步分格镜头，并逐渐揭示它们。

有一个常见的误区，有人将绘制分镜完全等同于设计一份精美的漫画。我认为这就是为什么很多从业人员并不乐意去绘制一份分镜的原因所在，他们畏惧去画一份精美的漫画；畏惧将一份精美的漫画实现为优雅的动画；畏惧花费大量的创造精力在计划过程上。总之，就是他们认为应该赶紧开始项目的工作，以尽快完成项目。我完全理解这种心理。为了避免有些人产生这种畏惧心理，建议你们忘掉心中对分镜不切实际的理解。

我是斯坦福大学和菲尔德自然博物馆（世界级自然博物馆，*http://www.fieldmuseum.org/*）聘用的科学插画家。我在大学同时也专攻美术专业。你们猜猜我所画的分镜是什么样的？我打赌一定会让你大吃一惊。 图 17-3 就是我画的一幅分镜。

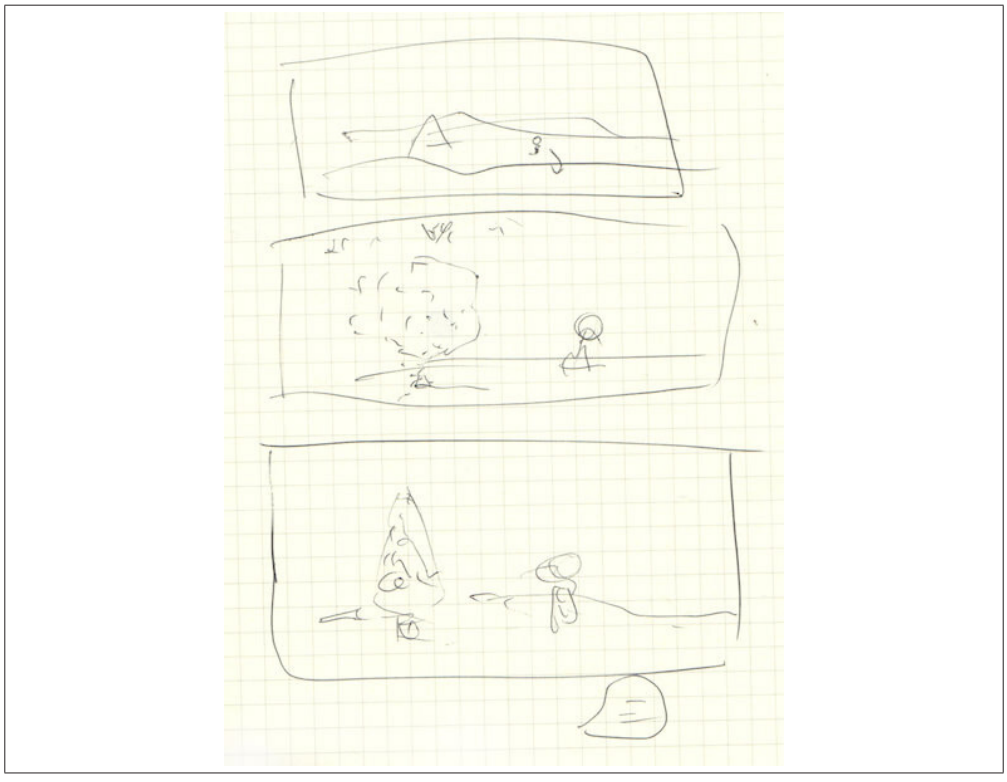


图17-3：丑陋但很有实用性的分镜

我会对如此简陋的初稿感到羞愧吗？一点也不会，这个初稿只花了我 45 秒，它是对我未来工作的一个整体性的把控和理解。它能够帮助我勾勒出未来我要花好几天绘制的动画的一些细节，没有它，我就无法做到事半功倍。分镜因人和使用场景而异。但是你没有必要像 Rachel Nabors（*http://bit.ly/2lZ55v5*）一样画得那么好，只需要草草地勾勒出草图就行了。 207

让我们重温一下有关用户共鸣的讨论。 现在你可以通过分镜来完成所有有关用户的交互。 你可以在动画分镜中就声明出从始至终的与用户的所有交互，可以考虑一下 James Buckhouse 的 *Story Map*（*http://bit.ly/2lPcPOm*）这篇文章。正如 James Buckhouse 在文章中所介绍的 story map，它是指一种新型的设计文档，其可以将所有相关的产品体验都画在一份文档中，能让设计人员一目了然，且脉络清晰。story map 就是用到了分镜的手法，它允许你以用户的身份从头到尾体验网站的全部功能。实际上这是分镜的进化版。这样的优势使你可基于预期目标和产出来做出一些有针对性的建议。

这也许不是你第一次听到有关动画中分镜的概念了：绘制多个粗糙的分格漫画，用来帮

助动画师来一个一个场景地分析并完成整个动画。但是你知道分镜的过程中也是需要进行配色的吗？就和你为网站设置配色和全套的商标一样。迪士尼和其他工作室，在制作分镜的时候也会为其中主要的任务和场景进行配色

208 这意味着你需要学会使用或者自己手工制作配色工具，例如 Adobe Kuler（*https://color.adobe.com/*）。这也许会花费一些时间，但是在你真正工作的时候却能节省大量的时间。我们都知道优化颜色选取工作是有意义的，用颜色选取工具会比 CSS 预处理器更加简单方便，使用它们可以更好地发挥你的优势。

工具

首先，我们需要引入一个草图（Thumbnails）的概念，草图和上文提到的分镜很像，但是使用场景不同，为了避免读者混淆，我们换了一个名字。

草图是你自己所做的简单的注释（参见图 17-4），草图并不一定通俗易懂，有时候时间长了你自己也可能会读不懂。但是它的作用是在极短的时间内，帮助你快速迭代思维和激发灵感。

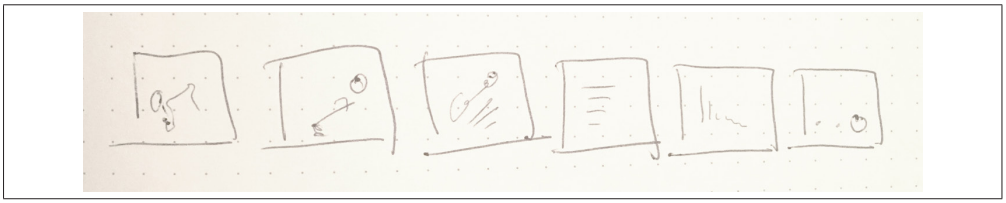


图17-4：草图是一种十分粗略的注释图解，用于帮助你在极短的时间内快速激发灵感

分镜是草图的一种升级版，它比草图绘制得更加逼真，也更容易让人辨识（参见图 17-5）。虽然它们也是很粗略的，但是可以用来展示很多场景，所以你可以用它们和同事交流。但是在正式的场合下，无论是分镜还是草图都不适合用于向他人做介绍。

209 而原型（Prototypes）相对于分镜和草图来说就显得更加高级了，它是你的最终成品的一个雏形，它可以表达出具体的交互逻辑。一个较低辨识度的原型一般是由一些基本图形组成的，并且制成初步的动画用来表现出交互逻辑（参见图 17-6）。正如你所见，原型只是部分功能的展现，所以你不需要把整个网站的全部功能都复现在一个原型内。

210 这里有一篇 Yaroslav Zubko 写的文章（*http://bit.ly/2mnbuBr*），其中专门介绍了如何使用基础图形制作原型，还包含了大量的 GIF 实例（*http://bit.ly/2jhdABm*）。

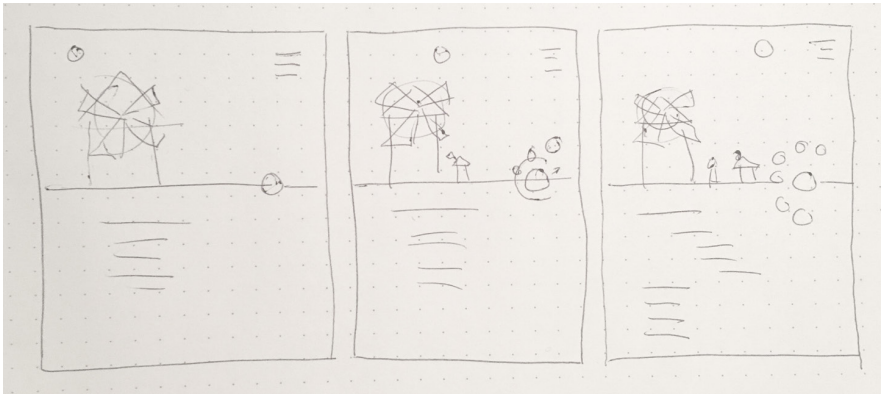


图17-5：分镜相比于草图更易让人辨识，但是它们都不是完整的画作，你可以和同事探讨灵感的时候使用分镜，即使它们确实画得不怎么好

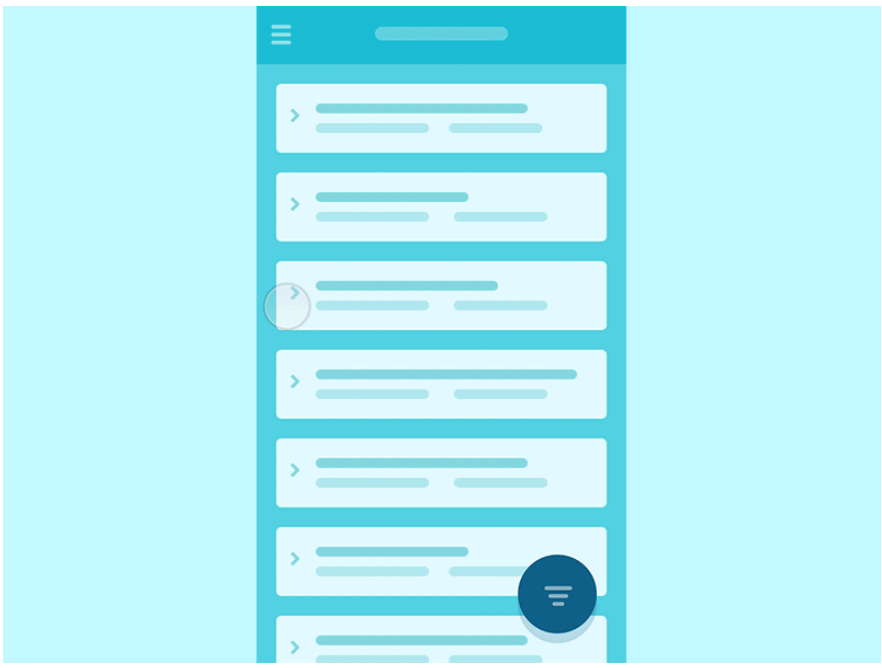


图17-6：Yaroslav Zubko 以基础图形制作的 GIF 格式的UI原型

还有一种工作方式是将现有网页和扁平化模型截图，然后将截图设置为背景图片，将需要添加的新动画元素以绝对定位的方式设置动画。这种方法适用于为他人做演示的时候，这可以快速获取元素并让元素运动起来展示你的交互方案。这种方式不会花费你太多的

时间。并且，它看起来和最终效果十分贴近，很适合给股东做演示。

如果你还困惑于设计原型时如何协调设计和编码之间的工作，这里为大家展示一些很棒的工具可用于制作可用的原型。

- Principle：一款可以将原型动画化的工具（*http://principleformac.com/*）。
- FramerJS：一款用于绘制图形、设计原型、展示给同事的工具，还可以使用编码的方式（*https://framer.com/*）。
- After Effects：大名鼎鼎的 Adobe 出品的，它可以为电影、电视、视频添加可用的动画（*http://www.adobe.com/products/aftereffects.html*）。
- Keynote：苹果出品的演示工具，能够制作出绚丽夺目的效果（*http://www.apple.com/keynote/*）。
- Straight-up code：可直接手写效果代码。

211 就我个人来说，我更倾向于自己手写代码。一来我觉得以上这些工具需要我花时间去学习，二来手写代码可以提升自身的开发能力。这样还可以使自己在原型设计方面理解得更加深刻，同时在实际开发的时候也不需要在交互动画这部分花时间，可以直接使用原型中的交互动画代码。

杀死汝爱

杀死汝爱（Murder your darlings）是英国作家亚瑟·奎勒·库奇（Arthur Quiller-Couch，*http://bit.ly/2m3Mi1X*）说的一句话。这句话同样适用于设计界——不要害怕推翻或修改某些东西，即使自己看起来觉得做得比较完美，你永远都不可能一次就把东西做好，所以，请学会放松和犯错。无论你是一名设计师、开发者，或两者兼顾，一开始并不会有太好的锻炼机会。直到你不断地产出许多丑陋的作品、尝试写过许多不同时长的动画，并且也搞砸过很多动画作品之后，你才能慢慢上道。学好任何东西都需要一个积累的过程，试问你学习 JavaScript 编程难道只学习使用一种框架吗？并不是吧？难道你学习设计是使用一种构图方法吗？我想恐怕也不是吧？同样的法则也适用于学习动画。

试想一开始你可以同时打开图形编辑工具和文本编辑工具。在动画时间轴上顺畅地向前向后进行滑动操作调试，不要害怕去回忆每一个步骤，或是优化动画和修改代码。你要准备好完成任务所需的工具，例如前面提过的 SVG 优化工具（*http://bit.ly/2lPUOkp*），借助工具的力量你可以快速完成工作。再往后，你就可以逐步丢开这些工具。懒惰和马虎的你逐渐会学会如何调试、修改代码，或者重建之前所做的图片和所写的代码，使它们成为你真正需要的作品。

你将会多次调整动画各部分的延时和缓动函数。依我看，我觉得这一阶段最简单的

方法是借助工具，例如使用 GreenSock 所提供的 TimelineLite（*http://greensock.com/timelinelite*）调整动画片段。它可以帮助你对动画元素进行交错排列、重叠等。

CSS 对于小型的 UI 交互设计来说是非常有用的。事实上，我建议你使用 CSS 来做交互实例，因为这样不需要加载其他资源。但是如果你需要制作有超过两个动画的场景，就应该考虑选择使用 GSAP 了。GSAP 能够前后调整动画帧，或者设置多个元素同时运动。这是一个非常强大的工具，尤其是你需要排练和调整动画的时候。如果你的动画的初始动画帧延时有些变化，那么就需要重新计算后面全部的动画帧，但使用 GSAP 的时间轴就不必这么做了。

正如上文所言，如果延时没有变化，那么没有什么大问题。不知道你注意到没有，有些元素并没有运动。这样会让观众的大脑产生错觉，认为动画已经暂停了。延时对于动漫、幽默剧来说是非常重要的。同样对于 UI 动画来说，如果想让 UI 动画表现出自然和无缝感，也必须控制好动画的延时。

正如在所有的设计中，动画的部分看起来比较简单也毫不费力，但是实际上有时候是最难完成的。 212

设计和编码的工作流程

上文已经很清楚地讲述了分镜在动画的设计和计划阶段的重要性。同样，如果对编码过程进行良好的设计和计划也会对整个工作流程有益。如果你能够以同设计稿一样的逻辑来组织代码，就可以获得一个清晰易懂的代码结构，并且可以很好地反映设计思路，这样开发者就可以互相参照着实现两者。

代码中的函数名应该能直接对应你所设置的场景信息，例如，sceneOne 这样的命名方式虽然看起来不错，但是一旦项目庞大且复杂起来，需要长期维护的时候，就会限制你和你的团队的工作效率，因为这样的命名方式并不直观，无法让人一眼就知道这个场景是用来干什么的。所以你需要根据它们的特征来设计函数名称，同时以清晰的方式编写可以反映你的设计的代码，这意味着减少了许多关于作用域、JS 和 Sass 变量指派上存在分歧的问题。同时，当开发完后需要回去调整代码的时候，一个清晰的设计结构也有助于你再次找到需要进行调整的位置，并且知道接下来的逻辑是什么。

制作动画组件库

有时候人们不知道如何将多种动画效果合并到动画库。所有的动画属性都能够以类的形式划分，这样就允许程序开发人员和设计人员根据自己的需求快速组合和迭代所需要的效果，只要能够保证语法正确即可。这里有一个以 Sass 为基础写的 CSS 动画组件样板（代

码有点长，请见谅）：

```
// ---- timing ----//
$class-slug: t !default;

@for $i from 1 through 7 {
    .#{$class-slug}-#{$i} {
        animation-duration: 0.8 - (0.1s * $i);
    }
}

// ---- ease ----//
$easein-quad: cubic-bezier(0.55, 0.085, 0.68, 0.53);
$easeout-quad: cubic-bezier(0.25, 0.46, 0.45, 0.94);
$easein-back: cubic-bezier(.57, .07, .6, 1.71);
$easeout-back: cubic-bezier(0.175, 0.885, 0.32, 1.275);

.entrance {
    animation-timing-function: $easeout-quad;
}
.entrance-emphasis {
    animation-timing-function: $easeout-back;
}
.exit {
    animation-timing-function: $easein-quad;
}
.exit-emphasis {
    animation-timing-function: $easein-back;
}

// 我们可以创造一个类，并使用 @extended 来继承类之间共同的属性
// .anim-fill-both 拥有 .pop 和 .unpop 共同的属性
.anim-fill-both {
    animation-fill-mode: both;
}

// animations
@keyframes pop {
  0% {
    transform: scale(0.9) translateZ(0);
  }
  100% {
    transform: scale(1) translateZ(0);
  }
}
```

213

```
.pop {
    animation-name: pop;
    // 使用 extend 继承共有属性
    @extend .anim-fill-both;
}

@keyframes unpop {
    0% {
        transform: scale(1) translateZ(0);
    }
    100% {
        transform: scale(0.9) translateZ(0);
    }
}

.unpop {
    animation-name: unpop;
    @extend .anim-fill-both;
}
```

你也可以直接看线上的例子（*http://codepen.io/sdras/pen/qqVrxy*）。

例子中我创建了一些时间单元，和 h1、h2、h3 一样是递增的关系。我把这些时间单元叫作 t1、t2、t3 等。t1 使动画延时最长，而 t5 和 h5 一样是最小的，能够给元素带来最少的动画延时（通常在 0.25 秒左右）。我使用 .entrance（进入）、.exit（退出）、.entrance-emphasis（增强式进入）、.exit-emphasis（增强式退出）这些大家通识的类名来保存动画的缓动函数。在上面的例子中，只有控制缓动函数和 fill-mode 的两种动画类是可以通用的。我们对不同的动画使用 animation-name 来定义不同的动画关键帧。当需要绘制大量的动画关键帧时，我建议你尝试先制作 5 到 6 条来检验你是否真的需要绘制那么多动画帧（上文中的例子并不需要绘制大量的关键帧，所以只写了两条）。如果动画库中有 30 多种不同的动画效果，那么这个动画库可以称得上是较优秀的动画库了。但是，动画库就好像你的调色盘一样，调色盘中颜色多了会给人一种凌乱的感觉。动画库中如果充斥着太多不是很必要的代码，请带着批判的方式思考这样创造出来的动画是否真的是你需要的。

214

虽然上面的例子只是一个非常简单的版本，但是如果是一个成熟稳健的动画系统的话，一定会有很多动画组件是可以复用的（通过类）。这样可以节省很多迭代和原型制作的时间，并且还易于在同一个动画中调整不同种类的动作感觉（通过切换不同的类）。

如何写一个成功的动画组件？我们可以先试着做一个简单而成功的对话框组件（dialog）。在一个大型网站中，你会看到有一种对话框会在不同的地方被多次使用，所以写一个类

似的组件能够让你真正地将精力集中在这上面，让你的制作水平能够快速提升。但是你需要避免一些问题：

- 不要在一段时间内连续弹出同样内容的对话框。
- 不要使用 GIF 图，因为 GIF 图体积较大，耗流量，对移动端不友好，且在视网膜高清屏上会显得十分模糊。

同时，你也可以将一个组件划分成多个精致的小组件，这样允许组件多次复用，可提高效率。

React 和 Vue 可以完美地实现组件的多次复用，所以你可以创建一些通用动画组件，这些组件允许你在不同的地方使用。记得利用好 React 和 Vue 的框架优势，例如，和上文中提到的例子一样，使用 props 参数调整不同种类的动画延时和缓动函数（上面的例子使用类进行分类调整）。

权衡动画开发的优先级

有些时候，一些开发者并不知道如何简单地设计动画，因为他们不知道如何取舍功能的优先级。如何合理地设计系统也是我们曾经探索过的问题。在 2016 年的 CSS 开发者大会上，Rachel Nabors（*http://rachelnabors.com/*）为我们展示了动画方面需求的优先级（参见图 17-7 和图 17-8），大家可以按照她所规划的优先级开发自己动画的代码库（转载需要得到 Rachel Nabors 的同意）。

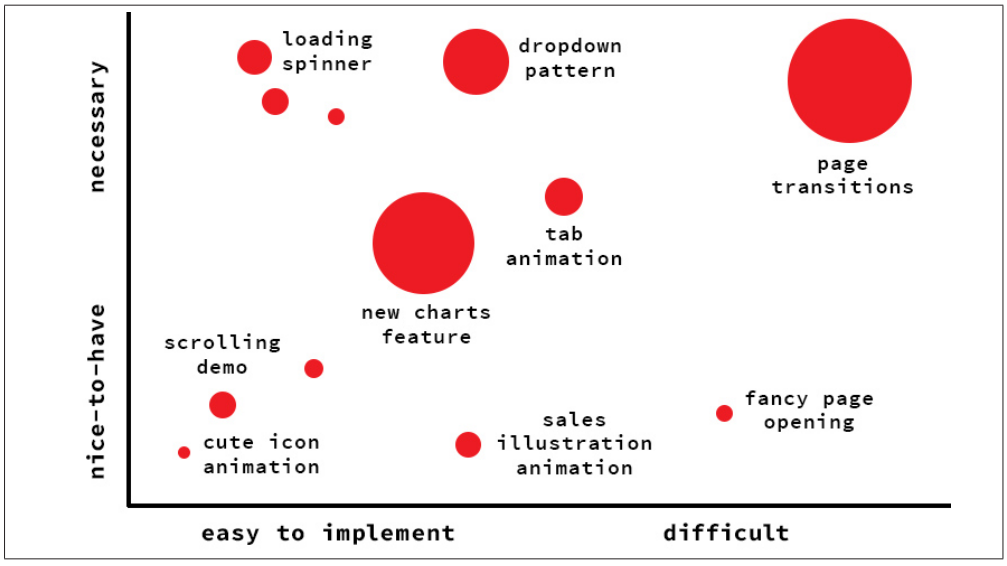


图17-7：想完成的需求和用户真正需求进行比较

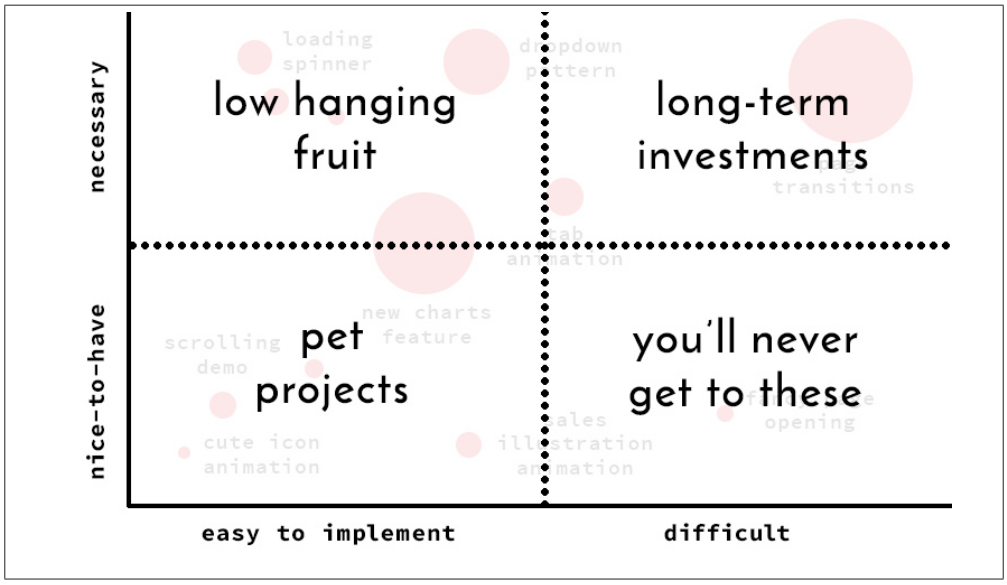


图17-8：承接上图，不同需求的完成难度

图 17-7 和图 17-8 提供了一些建议，指出了在实际开发过程中哪些是相对必要的工作，请带着批判性的思维去理解她所说的话。原则上只要你能保证你的工作的产出是用户真正需要的，而且可以复用，那么你就可以去开发这些动画。 215 216

好的公司都流行这么一种准则：对于一些不太能创造经济效益的工作，例如每个网站中的“关于我们”，我们不会尽全力开发一些用户所期望的炫酷的页面。但是我们可以让用户知道，在联系人的邮箱地址的展示效果上，是会有一点小小的动画过程的，并且最终可以产生一个不错的提示效果。

成功地制作一个小项目有助于提升团队的信任感，并且让你的合作伙伴了解你们是一支怎样的团队。建立一种良好的关系可以促进以后更复杂的项目合作。但是，不需要过分投入这种小项目，良好的沟通才是合作的关键。

时间就是金钱

在网站开发过程中，动画往往被认作为是一个事后点缀的功能，我们一般先设计网站原型，开发实际网站，在一切都完成后才添加动画。就因为如此，动画组件往往被人们形象地比喻为奶油蛋糕上的点缀物（草莓或巧克力）。动画开发只有在整个网站开发完成的基础上，经过分镜和原型设计后才能进行开发。开发完成后会对网站的用户体验有一个高性能和实质性的提升。

Active Theory 是一个专注于制作卓越品质动画的工作室，他们只服务于大型客户群。他们在产品设计过程中会积极地和用户交流，让用户了解他们正在设计的，能够给人带来惊奇的 Web 体验项目。

效仿 Active Theory，我们应该做哪些改变呢？比如，销售人员的工作目的是增加投资回报率；而程序员就是提升产品质量，造出一些有用的、能够吸引用户、增强用户体验的轮子。虽然这样会增加时间和人力投入，但是，会获得相对应的回报。相反，如果每次只是投入一点生产力，长期下来，其实最后所得的回报率不会太多。如果想了解如何和客户进行有效沟通的知识，可以看看由 Mike Monteiro 写的 *You're My Favorite Client* 或 *Design Is a Job*。

在为已成型的项目添加点缀的动效之前，我们还需要做一些工作来保证所做工作的有效性。

首先，我们需要和用户进行有效的沟通，这并不意味着要强制用户接受我们的想法。我们需要做的是为用户解释项目的预期收入，对做出的产品进行 A/B 测试（*http://bit.ly/2mDMZvS*）后，以得出的数据作为根据，来证明我们做法的合理性。同时，为用户节省大量的时间成本。

为了让我们的工作更简单，当没有多余时间搭建一个完整的效果时，可以用以下两种方法为用户展现我们的设计原型。第一种方式是可以在 CodePen（*http://codepen.io/patterns*）上收集他人写好的组件（按钮、进度条等），拼凑起来给用户预览。第二种方式，就是上文提到的，将现有网页和扁平化模型截图，然后将截图设置为背景图片，将需要添加的新动画元素以绝对定位的方式设置为动画来给用户预览。CodePen 虽然是一个很好的设计平台，拥有很多资源，但要注意，这些新颖的组件实际上并没有经过实战和市场调研。所以，你需要自己进行衡量，例如使用 A/B 测试（将两组或多组 UI/UX 样式同时上线，给多组用户使用，选取反馈情况好的那一组）来证明哪一种动画组件的性价比更好。实际的数字通常比项目经理的主观意见更具说服力。

217

一旦你在这方面取得了进步，可以试着按照下面列出的从高到低的优先级来重新计划一下你的开发计划：

- 开发时间
- 用户体验
- 性能
- 配色
- 内容组成
- 用户浏览所需要花费的时间

要做好这些看起来确实有些复杂。但是经过以上的学习，你应该可以在几秒内给自己的项目在这几方面一个大致的评分，这也是你能力提升过程中必须经过的坎儿。如果你依然缺乏经验，那么需要多历练。如果你的网站资源加载时间过长，那么就需要细心地优化你的资源加载（包括图片、SVG、JavaScript 脚本、动画库）。如果你的网站已经有多种配色方案了，那么你需要学会在组件中复用那些颜色。

其他方面的限制

Frank Thomas 和 Ollie Johnston 在他们的书 *Disney Animation: The Illusion of Life* 的开头引用了一句我一直都很喜欢的 Walt Disney 说过的一句话：动画可以将我们心中所幻想的世界描绘出来。这句话确实很准确地定义了动画的力量：在动画中，你可以创造任何可能发生的事，你可以创造一个世界、毁灭一个世界、激起观众兴趣、讽刺某些事物。

制作动画时需要在很多方面进行考虑。如果只考虑用户体验，而没有考虑其他方面，那么动画依然不会为观众带来好的体验。除了正常的设计之外，我们同样需要花大量时间去考查各种需要考虑的因素，这么做的回报就是：借助这些（知识或调研）准备，能够描述出产品（网站）特性，这些特性能够帮助我们规划好整体风格，从而实现用户界面。

索引†

†： 索引所列页码为本书英文版页码，请参照正文侧边用“▭”表示的原书页码。

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z

关于作者

Sarah Drasner 是一位优秀的布道者、技术顾问。Sarah 和 Val Head 是 Web 动画工作室的联合创始人。她被授予高级 SVG 动画前端之师之称，并曾担任 Truli（Zillow）的 UX 设计师和经理。Sarah 赢得了一系列奖项，包括 CSS Dev Conf 的“最佳设计奖”，以及来自 CSS 设计奖的“最佳代码辩论者”。作为一名 Web 开发人员和设计师，她已经有 15 年之久的工作经验，她还曾担任过插画师和大学教授，并在 Santorini 教过一个 Byzantine 的图标画师。

封面介绍

本书封面上的动物是白梢冠蕉鹃（Knysna turaco，其学名叫作 Tauraco corythaix），属于蕉鹃科（因喜欢吃香蕉，因而又被称为 banana-eaters）的一种，分布在南非和斯威士兰的丛林中。

由于本身的特征以及羽毛的颜色，所以白梢冠蕉鹃在鸟群中有很高的辨识度。它们可以长到 15 至 17 英寸（包括长尾巴），大多是绿色的，这有助于它们与树木融为一体。另外，它们拥有红颜色的羽毛，在翅膀的顶部能够看到蓝色的阴影。需要说明的是，鸟喙的颜色是亮红色中掺杂着淡橘色，这种颜色接近于它们眼睛的颜色，在眼睛底部以及眼睛顶部的部分位置，都内衬着白色眼影。除雏鸟之外，绿色的羽冠也都被冠上白色的条纹。

像其他外来鸟一样，白梢冠蕉鹃吃的食物过度依赖于昆虫、水果以及蚯蚓。不过还好，吃的食物比较充足，所以白梢冠蕉鹃的种群数量才能够保持稳定。

出于繁殖的目的，白梢冠蕉鹃会在树上搭建浅巢（shallow nests）。至于哪些月份是它们的繁殖季节，这因地区而异。在繁殖的那段时间，它们会产下一到两枚蛋，并且会由雄鸟和雌鸟共同孵化，该过程会持续 12 到 21 天。需要说明的是，两枚蛋只孵化出一只雏鸟，这并不稀奇。雏鸟出生 18 天之后就可以离开鸟巢独自觅食了。不过此时的它们，还不够成熟、独立，要成为成年鸟，还需要 3 个星期的时间。

O’Reilly 出版的图书封面上的动物，很多都是处于濒临灭绝的动物，它们存在的重要性不言而喻。要了解更多有关如何救助这些濒危动物的知识，请访问 *animals.oreilly.com*。

这幅封面插图是卡伦·蒙哥马利（Karen Montgomery）根据 *Wood’s Illustrated Natural History* 的内容，采用黑白手法雕刻而成的。